

TP312C-62  
683  
1-

# C 语言完全手册

基本概念、函数参考、编程实例  
与试题集锦

杨 峰 编著

科学出版社

科学出版社  
PDG



## 内 容 简 介

本书从3个不同的角度深入浅出地向读者介绍了C语言的知识,帮助读者提高C程序的设计能力和C语言的应试能力。全书分为3部分共18章,内容涵盖C语言的基础知识、C库函数介绍、经典C编程实例与常见试题解析。本书的最大特点是内容全面、实用性强,既有知识介绍,又有实例解析。通过对本书的学习,可以使读者全面掌握C语言的基本知识,同时提高读者的编程能力和应试能力。

本书既可作为C语言初学者的实用教材,也可作为具有一定编程经验的程序设计人员的参考资料,同时也会为那些准备参加C语言面试、C语言等级考试及与C语言相关的其他考试的读者提供有益的帮助。

### 图书在版编目(CIP)数据

C语言完全手册:基本概念、函数参考、编程实例与试题  
集锦/杨峰编著—北京:科学出版社,2008

ISBN 978-7-03-022524-5

I. C… II. 杨… III. C语言—程序设计—技术手册  
IV. TP312-62

中国版本图书馆CIP数据核字(2008)第105220号

责任编辑:高莹 / 责任校对:李玉茹

责任印刷:科海 / 封面设计:林陶

科学出版社出版

北京东黄城根北街16号

邮政编码:100717

<http://www.sciencep.com>

北京市鑫山源印刷有限公司

科学出版社发行 各地新华书店经销

\*

2008年8月第一版

开本:16开

2008年8月第一次印刷

印张:28.25

印数:000 1~4 000

字数:687千字

定价:49.80元

(如有印装质量问题,我社负责调换)



# 前 言

C 语言作为一门面向过程的程序设计语言已经发展了近 30 年。然而在当今各种编程技术层出不穷的时代，C 语言不但没有像其他同时代诞生的编程语言那样被历史所淘汰，反而越发地受到人们的重视。这主要归功于 C 语言强大的功能以及其他编程语言无法比拟的优点。系统级的程序设计、嵌入式系统开发、工业自动化控制等领域都是 C 语言大显身手的舞台。

当前市面上有关 C 语言的教材种类繁多、琳琅满目。大致可分为 3 类：第 1 类是单纯的教材，介绍 C 语言的基础知识；第 2 类是 C 语言函数手册，介绍了大量的 C 函数；第 3 类就是一些有关 C 语言的编程实例、C 语言习题辅导等书籍。这 3 类书都从不同的角度向读者传授了 C 语言的知识，但也存在一些问题。

首先，知识结构的系统性无法保证。不同的书作者不同，对 C 语言的理解也不尽相同，讲授的方式也不一样，因此读者想要博览这 3 类书籍从而精通 C 语言程序设计是有一定难度的。其次，读者的学习量过大，面对着大量的书籍容易使人望而生畏。由于每类书籍中知识量的限制，作者往往在编写书籍时长篇累牍，因此不乏一些重复和赘述。特别是一些国外的书，内容“细致入微”，让人抓不到重点，对于初学者来说是比较难以接受的。再次，书的种类繁多也给读者带来了经济上的负担。

基于这几点不足，笔者编写了此书。本书旨在从一个全新的角度向读者传达 C 语言的知识。让读者觉得学习 C 语言并不是那么困难，学习 C 语言只需要一本书。

## 与其他书籍相比，本书有何特点

### 1. 结构清晰，知识全面

本书涵盖了市面上常见的几种 C 语言书籍的内容，知识更加全面，结构更加清晰，从不同的角度帮助读者更好地掌握 C 程序设计的知识。全书共分 3 个部分，第 1 部分是 C 语言基础知识，让读者夯实基础，掌握 C 语言的全部内容。第 2 部分是 C 库函数，对每个标准 C 函数都做了详细解释和例程说明，方便读者查询参考。第 3 部分是经典 C 编程实例与常见试题解析，从实战和应试的角度让读者加深对 C 语言的理解，巩固学到的知识。

### 2. 实例丰富，题材新颖

本书在每一章都融入了大量的实例分析，旨在帮助读者更好地掌握知识。学习一门程序设计语言，最为重要的就是会使用它，应用它编写出高质量的代码。因此通过实例讲解知识点比单调地讲理论更加有效，更有助于提高读者的能力。另外，本书的例题题材新颖、生动、有趣，特别是第 3 部分的经典 C 编程实例一章，既有趣味性，又有难度，完全体现了算法和数据结构等方面的知识，对于读者提高编程能力，产生编程兴趣等都有好处。本书相关的源代码文件可以从 <http://www.khp.com.cn> 网站免费下载。



### 3. 讲解通俗，深入浅出

本书力求用简单的语言、通俗的方式、形象的描述，向读者介绍复杂的知识内容。C 语言的知识本身并不高深，但细节之处千头万绪，因此更需要用平实的语言，通俗地讲解知识。本书同时使用图示来解释较复杂的知识点，目的是更加直观形象。

### 4. 条理清晰，删繁就简

本书在 C 语言基础知识部分的每章最后都设有小结与回顾，将全章的知识点脉络进行梳理，使读者从全局把握本章的内容，提纲挈领。另外本书尽量减少不必要的内容介绍，突出重点，避虚就实，减少读者的学习负担和经济负担。

## 本书包括的内容

本书内容分为 3 部分：第 1 部分为 C 语言基础知识；第 2 部分为 C 库函数；第 3 部分为经典 C 程序实例与常见试题解析。

- ✧ 第 1 部分共包括 8 章，主要介绍了 C 语言的基本知识，其中包括：概述、数据类型和运算符、基本语句、函数、预处理命令、数组与指针、结构体与联合、位运算。
- ✧ 第 2 部分共包括 7 章，主要介绍了基于 C89 标准的 C 库函数。首先通过对 15 个标准 C 头文件的介绍，使读者了解 C 标准库的知识，从宏观上把握 C 库函数的使用。然后对各类常用的函数进行讲解，其中包括：C 标准库介绍、I/O 函数、字符处理函数、字符串处理函数、数学函数、时间和日期函数、其他函数等。还介绍了一些常用的非标准函数。
- ✧ 第 3 部分共包括 3 章，首先介绍了一些有关结构化程序设计、算法和数据结构等概念，为后面的内容作好铺垫。然后将一些经典算法和结构化程序设计的思想应用到实例中。最后是 100 道常见的 C 语言试题解析，以提高读者的应试能力，进一步巩固已有的知识，拾遗补缺，使本书更具有实用价值。

本书既可作为初学者的实用教材，也可作为具有一定编程经验的程序设计人员查阅参考的资料，同时也会为那些准备参加 C 语言面试、C 语言等级考试及与 C 语言相关的其他考试的读者提供一些有益的帮助。

本书由杨峰组织编写，同时参与编写的还有王俊标、陈晨、高守传、郭瑞、周宇炜、蔡雪焄、陈杰、荣飞、郑林、张路平、项宇峰、罗皓菡、赵正坤、程明雷，在此一并表示感谢。

作者

2008 年 5 月



# 目 录

## 第 1 部分 C 语言基础知识

第 1 章 概述 .....	2
1.1 C语言的产生和发展.....	2
1.2 C语言的特点 .....	3
1.3 C程序的开发平台.....	3
1.3.1 下载Turbo C开发环境.....	4
1.3.2 运行Turbo C开发环境.....	4
1.3.3 环境配置.....	5
1.3.4 程序的编写 .....	7
1.3.5 源程序的编译.....	8
1.3.6 程序的链接.....	8
1.3.7 程序的运行 .....	9
1.3.8 保存源文件退出Turbo C环境 .....	9
1.4 解析最简单的C程序.....	10
1.5 本章小结与要点回顾 .....	11
第 2 章 数据类型和运算符 .....	13
2.1 常量与变量 .....	13
2.2 C语言中的关键字.....	15
2.3 C语言的基本数据类型.....	16
2.3.1 整型.....	17
2.3.2 浮点型.....	20
2.3.3 字符型.....	22
2.3.4 枚举类型.....	24
2.4 运算符 .....	26
2.4.1 算术运算符.....	26
2.4.2 关系运算符.....	28
2.4.3 逻辑运算符.....	29
2.4.4 条件运算符.....	31
2.4.5 赋值运算符.....	32
2.4.6 逗号运算符.....	32



2.4.7 求字节数运算符 .....	34
2.5 本章小结与要点回顾 .....	34
<b>第3章 基本语句 .....</b>	<b>38</b>
3.1 C语句概述 .....	38
3.2 C程序的结构 .....	39
3.2.1 顺序结构 .....	40
3.2.2 分支结构 .....	40
3.2.3 循环结构 .....	41
3.3 基本的赋值语句 .....	42
3.4 分支语句和循环语句 .....	44
3.5 if语句 .....	44
3.5.1 第一种形式的if语句 .....	44
3.5.2 第二种形式的if语句 .....	45
3.5.3 第三种形式的if语句 .....	45
3.5.4 三种if语句的程序举例 .....	46
3.5.5 有关if的一些说明 .....	48
3.5.6 if语句的嵌套 .....	48
3.6 switch语句 .....	51
3.6.1 switch语句的一般形式 .....	52
3.6.2 带有break语句的switch语句 .....	53
3.6.3 有关switch语句的一些说明 .....	53
3.7 for语句 .....	54
3.7.1 for语句的一般形式 .....	54
3.7.2 有关for语句的一些说明 .....	55
3.7.3 for语句程序举例 .....	56
3.8 while语句 .....	57
3.9 do-while语句 .....	58
3.10 goto语句 .....	60
3.11 循环的嵌套 .....	61
3.12 break语句 .....	62
3.13 continue语句 .....	62
3.14 本章程序举例 .....	63
3.15 本章小结与要点回顾 .....	66
<b>第4章 函数 .....</b>	<b>70</b>
4.1 函数概述 .....	70



4.2	函数的定义 .....	72
4.3	函数的调用 .....	73
4.4	函数的返回值及类型 .....	75
4.5	函数的参数及传递方式 .....	77
4.6	函数的嵌套调用 .....	78
4.7	函数的递归调用 .....	79
4.7.1	求n的阶乘n! .....	80
4.7.2	汉诺塔 (Hanoi) 问题 .....	81
4.8	局部变量和全局变量 .....	83
4.8.1	局部变量 .....	84
4.8.2	全局变量 .....	85
4.9	变量的存储类别 .....	88
4.9.1	动态存储变量和静态存储变量 .....	88
4.9.2	auto变量 .....	89
4.9.3	用static声明的局部变量 .....	89
4.9.4	register变量 .....	91
4.9.5	同一文件中用extern声明外部变量 .....	92
4.9.6	多个文件中用extern声明外部变量 .....	92
4.9.7	用static声明外部变量 .....	94
4.10	内部函数和外部函数 .....	95
4.10.1	内部函数 .....	95
4.10.2	外部函数 .....	95
4.11	本章小结与要点回顾 .....	96

## 第5章 预处理命令 .....

## 99

5.1	预处理命令概述 .....	99
5.2	宏定义及其分类 .....	100
5.3	不带参数的宏定义 .....	100
5.3.1	不带参数的宏定义的一般形式 .....	100
5.3.2	宏定义的嵌套 .....	102
5.3.3	宏定义的其他应用 .....	102
5.4	带参数的宏定义 .....	103
5.4.1	带参数的宏定义的一般形式 .....	103
5.4.2	带参数的宏定义与函数 .....	104
5.4.3	使用带参数的宏定义的注意事项 .....	105
5.5	文件包含 .....	107



5.5.1 文件包含命令的一般形式 .....	107
5.5.2 文件包含的特点 .....	108
5.6 条件编译 .....	110
5.6.1 条件编译命令的一般形式 .....	110
5.6.2 条件编译的应用 .....	112
5.7 本章小结与要点回顾 .....	115
<b>第 6 章 数组与指针 .....</b>	<b>117</b>
6.1 数组的概念 .....	117
6.2 一维数组 .....	117
6.2.1 一维数组的定义 .....	117
6.2.2 一维数组的元素 .....	119
6.2.3 一维数组的初始化 .....	120
6.2.4 一维数组举例 .....	121
6.3 二维数组 .....	123
6.3.1 二维数组的定义 .....	123
6.3.2 二维数组的元素 .....	124
6.3.3 二维数组的初始化 .....	125
6.3.4 二维数组举例 .....	126
6.4 指针的概念 .....	128
6.4.1 内存的地址 .....	128
6.4.2 指针和指针变量 .....	129
6.5 指针型变量的定义 .....	130
6.6 指针型变量的引用 .....	131
6.6.1 指针变量引用的方法 .....	131
6.6.2 指针应用举例 .....	133
6.7 指针作为函数参数 .....	135
6.7.1 引入 .....	135
6.7.2 指针变量的函数参数 .....	136
6.8 指向数组元素的指针 .....	139
6.9 用指针引用数组元素 .....	140
6.9.1 指针对数组元素的引用 .....	140
6.9.2 几点注意事项 .....	142
6.10 数组名作为函数的参数 .....	143
6.10.1 数组名参数 .....	143
6.10.2 应用举例 .....	146



6.11 二维数组的指针 .....	148
6.11.1 二维数组的地址 .....	148
6.11.2 指向二维数组的指针 .....	151
6.12 字符数组 .....	153
6.12.1 字符数组的定义和初始化 .....	153
6.12.2 字符串与字符串的结束标志 .....	154
6.12.3 字符数组的输入输出 .....	155
6.13 字符串指针 .....	157
6.13.1 用指针指向字符串 .....	157
6.13.2 字符串指针作为函数的参数 .....	158
6.14 指针与函数 .....	160
6.14.1 指向函数的指针 .....	160
6.14.2 返回指针的函数 .....	162
6.15 指针数组和指向指针的指针 .....	163
6.15.1 指针数组 .....	163
6.15.2 指向指针的指针 .....	165
6.16 void型指针 .....	167
6.17 本章小结与要点回顾 .....	168
<b>第7章 结构体与联合 .....</b>	<b>172</b>
7.1 结构体概述 .....	172
7.2 结构体变量的定义 .....	173
7.2.1 先定义类型后定义变量 .....	173
7.2.2 在定义结构体类型的同时定义变量 .....	174
7.2.3 直接定义结构体变量 .....	174
7.2.4 关于结构体类型的几点说明 .....	175
7.3 结构体变量的引用 .....	176
7.3.1 结构体成员的引用 .....	176
7.3.2 结构体变量的初始化 .....	178
7.4 结构体数组 .....	179
7.4.1 结构体数组的定义 .....	179
7.4.2 结构体数组的初始化 .....	180
7.4.3 结构体数组举例 .....	180
7.5 指向结构体的指针 .....	182
7.5.1 指向结构体变量的指针 .....	182
7.5.2 链表简介 .....	185



7.6	联合的概念及定义 .....	185
7.7	联合变量的引用 .....	187
7.8	使用联合的注意事项 .....	188
7.9	自定义类型 .....	188
7.10	本章小结与要点回顾 .....	189
<b>第 8 章</b>	<b>位运算 .....</b>	<b>192</b>
8.1	概述 .....	192
8.2	位运算符 .....	192
8.2.1	按位与运算 (&) .....	193
8.2.2	按位或运算 ( ) .....	194
8.2.3	按位异或运算 (^) .....	194
8.2.4	取反运算 (~) .....	196
8.2.5	左移运算 (<<) .....	196
8.2.6	右移运算 (>>) .....	198
8.3	位运算中的规则 .....	199
8.3.1	不同类型数据之间的位运算 .....	199
8.3.2	位运算赋值运算符 .....	199
8.4	位段 .....	200
8.4.1	位段的定义和位段变量的说明 .....	200
8.4.2	位段的应用举例 .....	201
8.4.3	位段的几点说明 .....	202
8.5	本章小结与要点回顾 .....	202

## 第 2 部分 C 库函数

<b>第 9 章</b>	<b>C 标准库介绍 .....</b>	<b>206</b>
9.1	诊断: <assert.h> .....	207
9.2	字符类别测试: <ctype.h> .....	207
9.3	错误处理: <errno.h> .....	208
9.4	浮点算术运算常量: <float.h> .....	208
9.5	整型常量: <limits.h> .....	209
9.6	地域环境: <locale.h> .....	209
9.7	数学函数: <math.h> .....	210
9.8	非局部跳转: <setjmp.h> .....	211
9.9	信号: <signal.h> .....	212



9.10 可变参数表: <stdarg.h> .....	214
9.11 公共定义: <stddef.h> .....	214
9.12 输入输出: <stdio.h> .....	215
9.13 实用函数: <stdlib.h> .....	216
9.14 字符串函数: <string.h> .....	217
9.15 日期与时间函数: <time.h> .....	217
<b>第 10 章 I/O 函数</b> .....	<b>219</b>
10.1 文件概述 .....	219
10.2 clearerr 复位错误标志函数 .....	220
10.3 fopen、fclose 文件的打开与关闭函数 .....	221
10.4 feof 检测文件结束符函数 .....	222
10.5 ferror 检测流上的错误函数 .....	223
10.6 fflush 清除文件缓冲区函数 .....	224
10.7 fgetc 从流中读取字符函数 .....	226
10.8 fgetpos 取得当前文件的句柄函数 .....	227
10.9 fgets 从流中读取字符串函数 .....	227
10.10 fprintf 格式化输出函数 .....	228
10.11 fputc 向流中输出字符函数 .....	230
10.12 fputs 向流中输出字符串函数 .....	231
10.13 fread 从流中读取字符串函数 .....	231
10.14 freopen 替换文件中数据流函数 .....	232
10.15 fscanf 格式化输入函数 .....	233
10.16 fseek 文件指针定位函数 .....	234
10.17 fsetpos 定位流上的文件指针函数 .....	235
10.18 ftell 返回当前文件指针位置函数 .....	236
10.19 fwrite 向文件写入数据函数 .....	237
10.20 getc 从流中读取字符函数 .....	238
10.21 getchar 从标准输入文件中读取字符函数 .....	239
10.22 gets 从标准输入文件中读取字符串函数 .....	240
10.23 perror 打印系统错误信息函数 .....	240
10.24 printf 产生格式化输出的函数 .....	241
10.25 putc 向指定流中输出字符函数 .....	242
10.26 putchar 向标准输出文件上输出字符 .....	243
10.27 puts 将字符串输出到终端函数 .....	243
10.28 remove 删除文件函数 .....	244



10.29	rename重命名文件函数.....	245
10.30	rewind重置文件指针函数.....	245
10.31	scanf格式化输入函数.....	246
10.32	setbuf、setvbuf指定文件流的缓冲区函数.....	247
10.33	sprintf向字符串写入格式化数据函数.....	248
10.34	sscanf从缓冲区中读取格式化字符串函数.....	249
10.35	tmpfile创建临时文件函数.....	250
10.36	tmpnam创建临时文件名函数.....	251
10.37	ungetc把字符退回到输入流函数.....	251
<b>第 11 章 字符处理函数.....</b>		<b>253</b>
11.1	isalnum检查字符是否是字母或数字.....	253
11.2	isalpha检查字符是否是字母.....	254
11.3	isascii检查字符是否是ASCII码.....	255
11.4	isctrl检查字符是否是控制字符.....	255
11.5	isdigit检查字符是否是数字字符.....	256
11.6	isxdigit检查字符是否是十六进制数字字符.....	257
11.7	isgraph检查字符是否是可打印字符（不含空格）.....	257
11.8	isprint检查字符是否是可打印字符（含空格）.....	258
11.9	ispunct检查字符是否是标点字符.....	259
11.10	islower检查字符是否是小写字母.....	259
11.11	isupper检查字符是否是大写字母.....	260
11.12	isspace检查字符是否是空格符.....	261
11.13	toascii将字符转换为ASCII码.....	262
11.14	tolower将大写字母转换为小写字母.....	262
11.15	toupper将小写字母转换为大写字母.....	263
<b>第 12 章 字符串处理函数.....</b>		<b>265</b>
12.1	strcat字符串连接函数.....	265
12.2	strncat字符串连接函数.....	266
12.3	strcmp字符串比较函数.....	267
12.4	strncmp字符串子串比较函数.....	268
12.5	strcpy字符串拷贝函数.....	269
12.6	strncpy字符串子串拷贝函数.....	270
12.7	strlen计算字符串长度函数.....	271
12.8	strchr字符串中字符首次匹配函数.....	272
12.9	strrchr字符串中字符末次匹配函数.....	273

12.10	strspn字符集匹配函数.....	274
12.11	strcspn字符集逆匹配函数.....	275
12.12	strpbrk字符集字符匹配函数.....	276
12.13	strstr字符串匹配函数.....	277
12.14	strtok字符串分隔函数.....	278
12.15	strtod字符串转换成双精度函数.....	280
12.16	strtol字符串转换成长整型函数.....	281
12.17	strtoul字符串转换成无符号长整型函数.....	282
12.18	strdup字符串新建拷贝函数.....	283
12.19	strset字符串设定函数.....	284
12.20	strrev字符串倒转函数.....	285
12.21	swab字符串交换字节函数.....	286
12.22	strlwr字符串小写转换函数.....	287
12.23	strupr字符串大写转换函数.....	288
12.24	strerror字符串错误信息函数.....	288
12.25	atoi字符串转整型的函数.....	289
12.26	atol字符串转长整型的函数.....	290
12.27	atof字符串转浮点型的函数.....	291
12.28	memcpy字符串拷贝函数.....	292
12.29	memmove字块移动函数.....	293
12.30	memcmp字符串比较函数.....	294
12.31	memchr字符搜索函数.....	295
12.32	memset字符加载函数.....	296
<b>第 13 章 数学函数.....</b>		<b>297</b>
13.1	abs、labs、fabs求绝对值函数.....	297
13.2	div、ldiv除法函数.....	298
13.3	ceil向上舍入函数.....	299
13.4	floor向下舍入函数.....	300
13.5	fmod求模函数.....	301
13.6	frexp分解浮点数函数.....	301
13.7	ldexp装载浮点数函数.....	302
13.8	modf分解双精度数函数.....	302
13.9	exp求e的x次幂函数.....	303
13.10	log、log10对数函数.....	303
13.11	hypot求直角三角形斜边长函数.....	304



13.12	pow、pow10指数函数.....	305
13.13	sqrt开平方函数.....	306
13.14	rand产生随机整数函数.....	306
13.15	srand设置随机时间的种子函数.....	307
13.16	sin正弦函数.....	308
13.17	asin反正弦函数.....	308
13.18	cos余弦函数.....	309
13.19	acos反余弦函数.....	310
13.20	tan正切函数.....	310
13.21	atan反正切函数.....	311
13.22	atan2反正切函数.....	311
13.23	sinh双曲正弦函数.....	312
13.24	cosh双曲余弦函数.....	312
13.25	tanh 双曲正切函数.....	313
<b>第 14 章 时间和日期函数.....</b>		<b>314</b>
14.1	clock测定运行时间函数.....	314
14.2	difftime计算时间差函数.....	315
14.3	mktime时间类型转换函数.....	315
14.4	time获取系统时间函数.....	317
14.5	asctime日期和时间转换函数.....	318
14.6	ctime时间转换函数.....	318
14.7	gmtime将日历时间转换为GMT.....	319
14.8	localtime把日期和时间转换为结构.....	320
<b>第 15 章 其他函数.....</b>		<b>321</b>
15.1	calloc分配主存储器函数.....	321
15.2	malloc动态分配内存函数.....	322
15.3	realloc重新分配主存函数.....	323
15.4	free释放内存函数.....	324
15.5	abort异常终止进程函数.....	324
15.6	exit正常终止进程函数.....	325
15.7	atexit注册终止函数.....	326
15.8	getenv获取环境变量.....	327
15.9	bsearch二分搜索函数.....	327
15.10	qsort快速排序函数.....	329



## 第 3 部分 经典 C 编程实例与常见试题解析

第 16 章 C 语言常用算法.....	332
16.1 结构化程序设计 .....	332
16.2 程序的灵魂——算法 .....	334
16.3 常用的数据结构 .....	337
16.4 顺序表 .....	338
16.4.1 顺序表的定义 .....	338
16.4.2 向顺序表中插入元素 .....	339
16.4.3 从顺序表中删除元素 .....	340
16.4.4 程序举例 .....	341
16.5 链表 .....	343
16.5.1 创建链表 .....	344
16.5.2 向链表中插入结点 .....	344
16.5.3 从链表中删除结点 .....	345
16.5.4 程序举例 .....	346
16.6 队列和栈 .....	348
16.7 树结构 .....	349
16.8 图结构 .....	351
第 17 章 经典 C 编程实例.....	354
17.1 打印特殊图案 .....	354
17.2 打印乘法口诀表 .....	356
17.3 计算100以内的素数 .....	357
17.4 判断回文数字 .....	358
17.5 计算最大公约数 .....	360
17.6 寻找阿姆斯特朗数 .....	361
17.7 歌德巴赫猜想的近似证明 .....	363
17.8 百钱买百鸡问题 .....	366
17.9 求 $\pi$ 的近似值 .....	367
17.10 爱因斯坦的阶梯问题 .....	371
17.11 可扩展的数列排序 .....	372
17.12 八皇后问题 .....	374
第 18 章 常见 C 语言试题解析.....	378
18.1 C程序设计的基础知识.....	378
18.2 顺序结构 .....	385



18.3 分支结构 .....	388
18.4 循环结构 .....	396
18.5 数组 .....	403
18.6 指针 .....	411
18.7 函数 .....	420
18.8 结构与联合 .....	426
18.9 位运算 .....	429
18.10 文件操作 .....	433

# 第 1 部分

## C 语言基础知识

作为一门面向过程的程序设计语言，C 语言有着其他高级语言所不具备的优点，因此越来越为人们所重视和普及。它被广泛地应用于应用软件、系统软件和嵌入式系统的开发等。

掌握 C 语言的关键是对 C 语言的基础知识做系统、全面的了解。本部分分 8 章介绍了 C 语言的基础知识，第 1 章：概述；第 2 章：数据类型和运算符；第 3 章：基本语句；第 4 章：函数；第 5 章：预处理命令；第 6 章：数组与指针；第 7 章：结构体与联合；第 8 章：位运算。通过本章的学习，读者可以全面掌握 C 语言的知识，为将来的 C 程序设计实践打下基础。

资源知识  
PDG



C 语言是目前国际上应用广泛, 且具有发展前途的一门计算机高级语言。许多软件的开发都是用 C 语言实现的。由于 C 语言既具有一般高级语言的语言简洁、结构化、语法限制不太严格等优点, 又具有低级语言的可以对硬件进行描述等特性, 因此, C 语言既可以用来开发系统软件, 又可以用来开发应用软件。

## 1.1 C 语言的产生和发展

C 语言的产生要追溯到 ALGOL 60 语言。ALGOL 60 是一种面向问题的高级语言, 它产生于 20 世纪 60 年代。但由于其自身的局限性, 它并不适用于编写系统程序。因此, 1963 年剑桥大学推出了 CPL (Combined Programming Language) 语言, 以弥补 ALGOL 60 的不足。但 CPL 仍然存在着致命的缺陷, 虽然较 ALGOL 60 更加接近硬件, 但规模较大, 难以实现。在 CPL 的基础上, 1967 年剑桥大学的 Martin Richards 对 CPL 进行了简化, 并推出了 BCPL (Basic Combined Programming Language) 语言。1970 年, 美国贝尔实验室又对 BCPL 语言进行了进一步的简化, 设计出了既简单, 又接近硬件的 B 语言 (BCPL 的第一个字母)。但是, 由于 B 语言过于简单, 功能十分有限, 因此美国贝尔实验室在 1972—1973 年间, 在 B 语言的基础上又设计出了 C 语言 (BCPL 的第二个字母)。C 语言的问世是计算机高级语言发展的一个里程碑。

C 语言既保持了 B 语言的简洁、精练、接近硬件等优点, 同时克服了 B 语言的过于简单、功能有限等缺点。1978 年由美国电话电报公司 (AT&T) 贝尔实验室正式发表了 C 语言。同时由 B.W.Kernighan 和 D.M.Ritchie 合著了著名的《THE C PROGRAMMING LANGUAGE》一书。这本书又被称为《K&R》。但是《K&R》也并未定义一个完整的 C 语言标准。后来由美国国家标准协会 (American National Standards Institute, ANSI) 在《K&R》基础上制定了一个完整的 C 语言标准, 并于 1983 年正式发表, 通常称之为标准 C (ANSI C)。1987 年, ANSI 又公布了新的 C 语言标准——87 ANSI C。1990 年, 国际标准化组织 (International Standard Organization, ISO) 接受 87 ANSI C 作为 ISO C 的标准。

目前, 广泛使用的 C 编译环境都以 87 ANSI C 为基础。最流行的 C 语言编译系统包括 Microsoft C、Turbo C、Quick C 等。不同版本的 C 编译环境略有差异。



## 1.2 C 语言的特点

C 语言发展至今经久不衰，且为许多程序开发者所青睐，这与它的诸多特点是分不开的。总结起来，C 语言的主要特点可以归纳为以下几点：

(1) 语言简洁、使用方便、易学易用。C 语言总共只有 32 个关键字，9 种控制语句，而且语法简单易懂，可读性强，接近自然语言。

(2) 运算符丰富。C 语言的运算符十分丰富，总共有 34 种运算符。这不但增加了表达式的种类，灵活使用各种运算符，还可以使运算功能大大增强，从而实现其他高级语言难以实现的运算。

(3) 数据类型丰富。C 语言中有整型、浮点型、字符型、指针型、结构体、联合、枚举等丰富的数据类型。通过这些数据类型就可以增强程序设计的灵活性，实现较复杂的数据结构，从而提高编程质量。

(4) 语法限制不太严格，编程自由度大。

(5) 程序设计结构化。C 语言是典型的面向过程的程序设计语言，因此它具有良好的结构化控制语句。此外，在 C 语言程序设计中，函数是基本的功能单位，这样便于实现程序设计的模块化。

(6) 具有位操作能力，能够直接访问物理地址。这是 C 语言较之其他高级语言的优势所在。这种特性使 C 语言介于一般高级语言与低级语言之间，既能够对硬件进行操作，又不像低级语言那样复杂。

(7) 代码质量高，程序的可移植性强。

以上简要地介绍了 C 语言的特点。通过进一步深入学习，读者会对这些特点有更加深入的理解。基于以上内容，我们可以理解为什么 C 语言能够得到如此广泛的应用和青睐。也正是由于 C 语言具有这些其他语言无法比拟的优点，使得许多大型软件甚至是操作系统都可以用 C 语言来编写开发。

## 1.3 C 程序的开发平台

C 语言的开发平台很多，前面已经介绍过，最流行的 C 语言编译系统包括 Microsoft C、Turbo C、Quick C 等。当然，也可以用当下比较流行的用于面向对象程序设计和可视化程序设计的集成开发平台 Visual C 进行 C 程序的开发。但是，无论使用哪种开发工具，它们都无一例外地要实现 ANSI C 标准。与此同时，不同版本的 C 编译环境之间也略有差异。本节主要针对 Turbo C 开发环境做简要的介绍。因为它界面简单、直观、易于上手操作，熟悉了 Turbo C 开发环境后，再去学习理解其他开发环境就会容易许多。

Turbo C 是在微型计算机上广泛使用的 C 语言开发工具。它是一个集程序的编辑、编译、链接和运行于一身的集成开发环境。而且界面友好、直观、操作方便，是开发 DOS 环境下 C 程序的理想开发工具。



Turbo C 的功能很多, 这里只介绍完成一个 C 程序的编辑、编译、链接、运行等步骤所涉及的 Turbo C 的使用, 其他的功能用户可以参考相关手册。

下面介绍应用 Turbo C 开发一个 C 应用程序的完整流程。

### 1.3.1 下载 Turbo C 开发环境

用户可以从以下网站免费下载 Turbo C 开发环境:

- ✧ <http://www.soft1001.com/>
- ✧ <http://www.onlinedown.net/>
- ✧ <http://download.pchome.net/>
- ✧ <http://www.winyes.com/>

这里假设用户已经下载了 Turbo C 开发环境, 并将它安装或复制到 E:\TC 目录下, 然后我们进入下一步, 运行 Turbo C 开发环境。

### 1.3.2 运行 Turbo C 开发环境

双击 TC 子目录下的可执行文件 TC.EXE, 或是在 DOS 相应的目录下输入 TC 命令

```
E:\TC>TC <回车>
```

就可以运行 Turbo C 环境。程序打开后, 屏幕上出现 Turbo C 集成开发环境, 如图 1-1 所示。

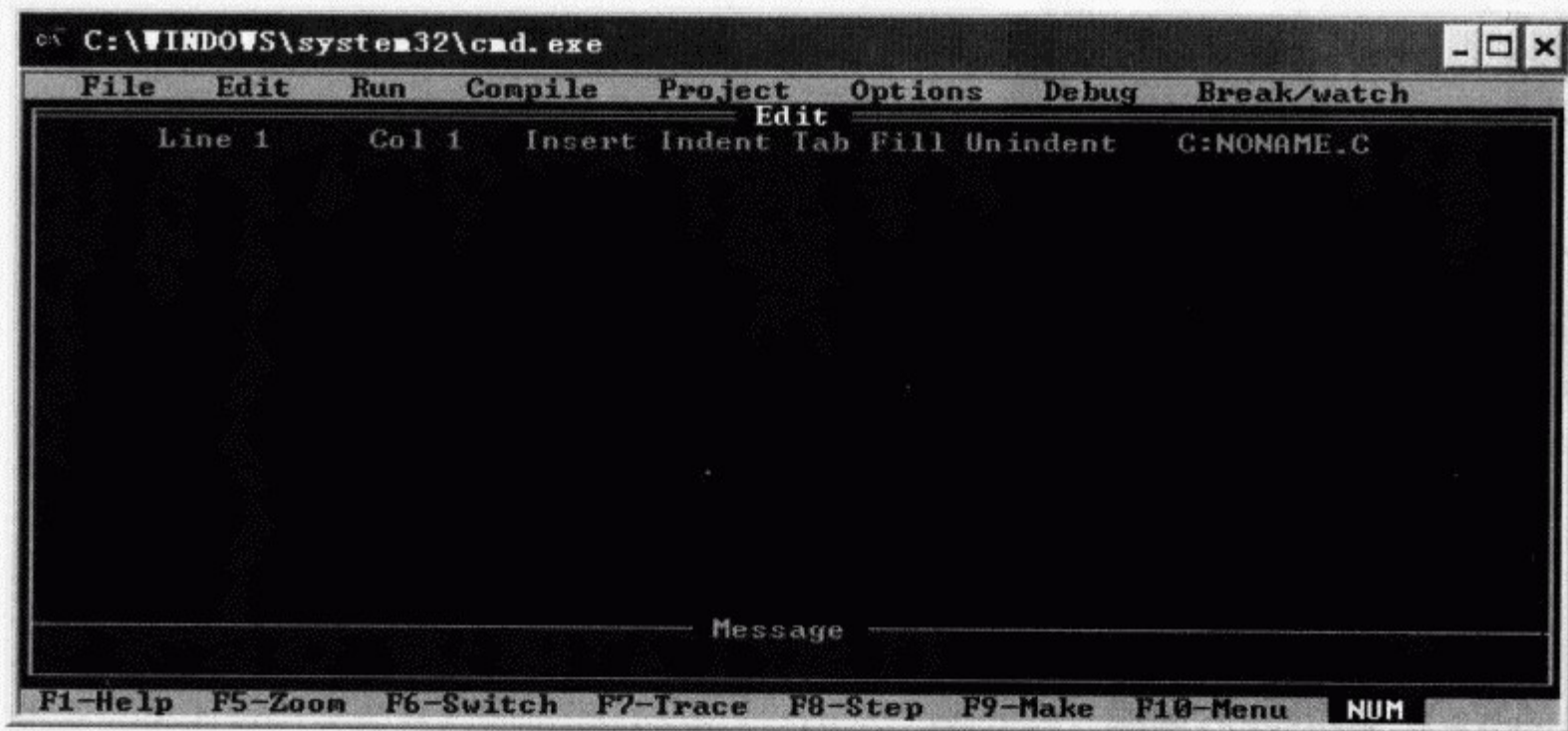


图 1-1 Turbo C 集成开发环境

可以看到, 集成环境的顶部是一行主菜单, 包含 8 个菜单项。通过主菜单提供的各种功能, 可以很好地实现程序的编辑、编译、链接、运行、调试、存取等功能, 实现人机的交互。这 8 个菜单项分别为 File (文件操作)、Edit (文本编辑)、Run (运行)、Compile (编译)、Project (项目文件)、Options (选项)、Debug (调试)、Break/watch (中断/观察)。要选中不同的菜单项, 只需在按下 Alt 键的同时按下要选择的菜单项的首字母即可。例如, 要选择 File 菜单项, 只需按下 Alt+F 组合键即可。



初次运行 Turbo C 环境，还要对 Turbo C 环境作一些必要的配置，以便今后可以方便正确地使用它来编写程序。

### 1.3.3 环境配置

有两个环境参数是用户必须了解的，也是用户可以修改以方便编程的。一是系统默认的 C 文件保存和打开路径；二是系统默认的 EXE 可执行文件输出路径。下面分别介绍。

#### 1. 系统默认的 C 文件保存和打开路径

C 语言的源程序文件是后缀为 .C 的 C 文件。Turbo C 有默认的 C 文件保存和打开路径。如果你的 TC 安装在 E:\TC 目录下，它的路径名一般是 E:\TC\PROJECT\。也就是说，用 Turbo C 打开一个 C 文件时，系统会默认地从 E:\TC\PROJECT\ 下寻找用户要打开的文件，而当用户保存 (Save) 一个文件时，也会默认地保存在这个文件目录下，如图 1-2 所示。

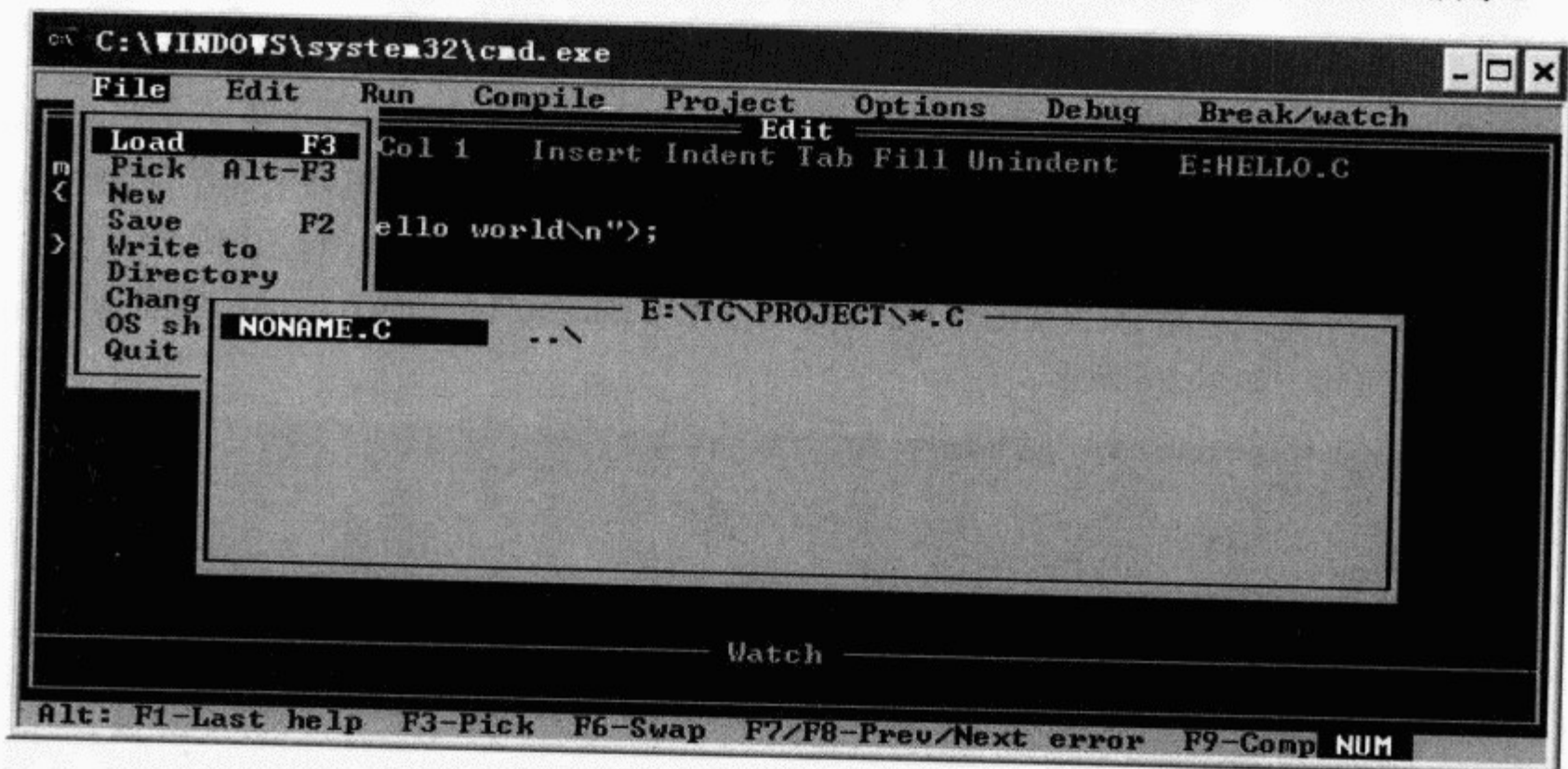


图 1-2 系统默认的 C 文件保存和打开路径

从图 1-2 中可以看出，当通过 File→Load 命令打开已有的 C 文件时，标题栏上显示的是 E:\TC\PROJECT\\*.C，也就是说系统默认地指向这个路径来寻找要打开的文件。同样，当用 File→Save 命令保存编辑好的文件时，系统也会默认地指向这个路径来保存 C 文件。

这个默认的路径是可以修改的，因为用户并不一定希望把所有编写好的文件都保存在系统指定的文件夹下，而是保存在别的文件夹下统一管理。可以通过 File→Change dir 命令来改变系统的默认路径，如图 1-3 所示。

如图 1-3 所示，只要按 Alt+F 组合键选中 File 菜单，再移动光标至 Change dir 菜单项，按 Enter 键，就会显示出 New Directory 窗口，然后通过键盘修改默认的路径。例如图中的 E:\TC\my。

修改好了系统的默认路径，再打开 C 文件或保存 C 文件时，系统就会默认地指向这个路径进行操作。



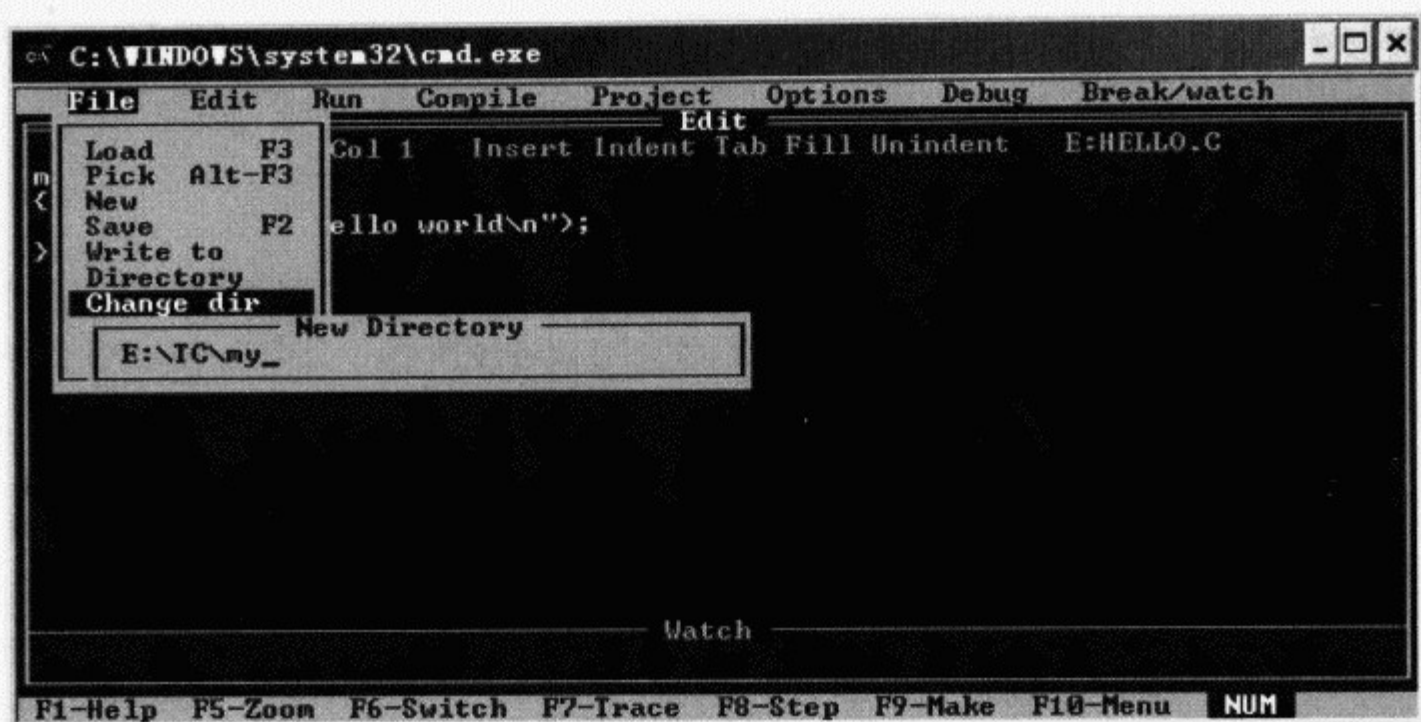


图 1-3 修改默认的 C 文件保存和打开路径

## 2. 系统默认的 EXE 可执行文件输出路径

C 程序编译和链接之后会生成可执行文件\*.EXE，默认情况下，系统会将生成的可执行文件输出到当前文件夹下的 OUTPUT 目录下。也就是说，对于本例，每当编译链接好一个 C 程序，系统就会在 E:\TC\OUTPUT 下生成一个可执行文件。当然，这个默认的路径也是可以修改的，如图 1-4 所示。

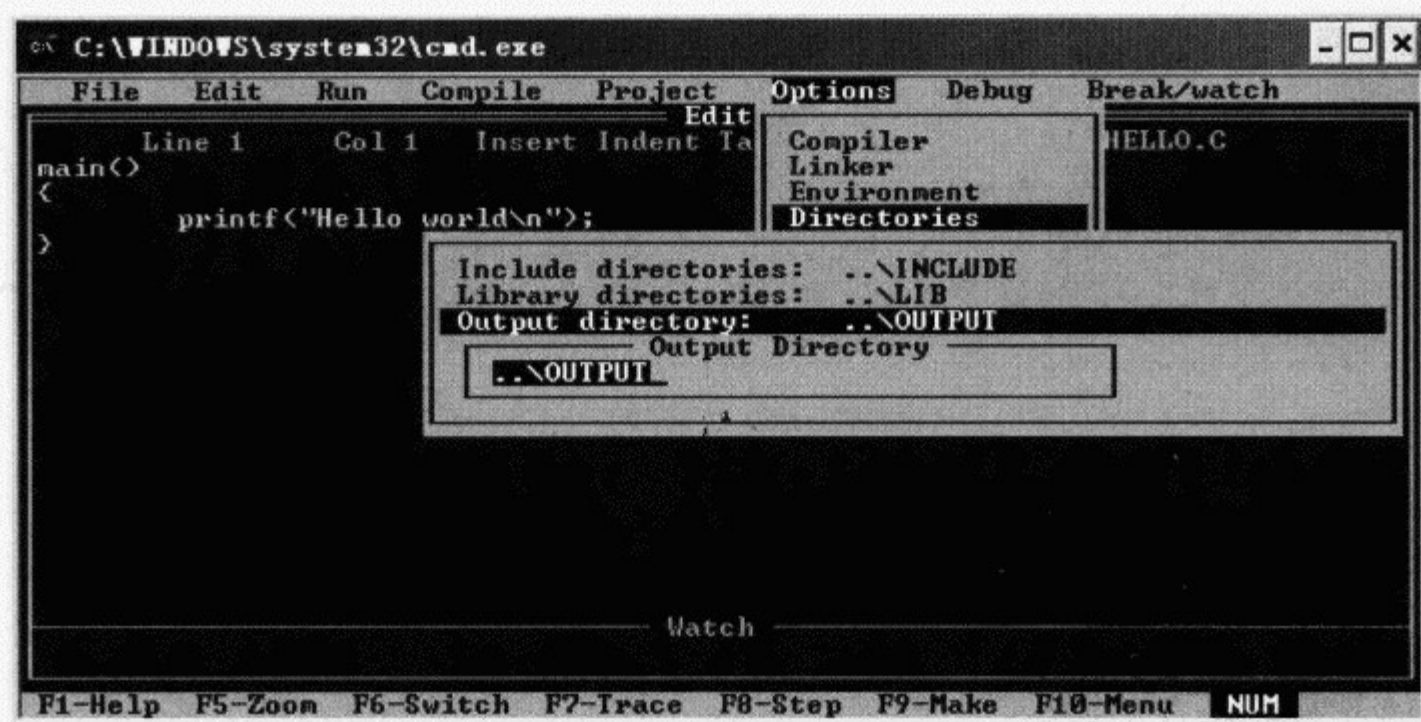


图 1-4 修改系统默认的 EXE 可执行文件输出路径

如图 1-4 所示，只要按 Alt+O 组合键选中 Options 菜单，再移动光标至 Directories 菜单项，按 Enter 键，再移动光标至 Output directory 选项，按 Enter 键，就会显示 Output Directory 窗口，然后通过键盘修改默认的路径，例如修改为 E:\TC\PRO。这样再生成的可执行文件就会被输出到 E:\TC\PRO 目录下。

配置好环境，接下来就可以进入编写程序这个环节。



### 1.3.4 程序的编写

接下来就要开始编写 C 程序了。按 Alt+E 组合键进入编辑状态，只要在 Edit（编辑）状态下，就可以通过键盘和光标编辑修改源程序。例如在编辑区编写如图 1-5 所示一段最简单的 C 程序，它的作用是在屏幕上打印“Hello world”字符串。

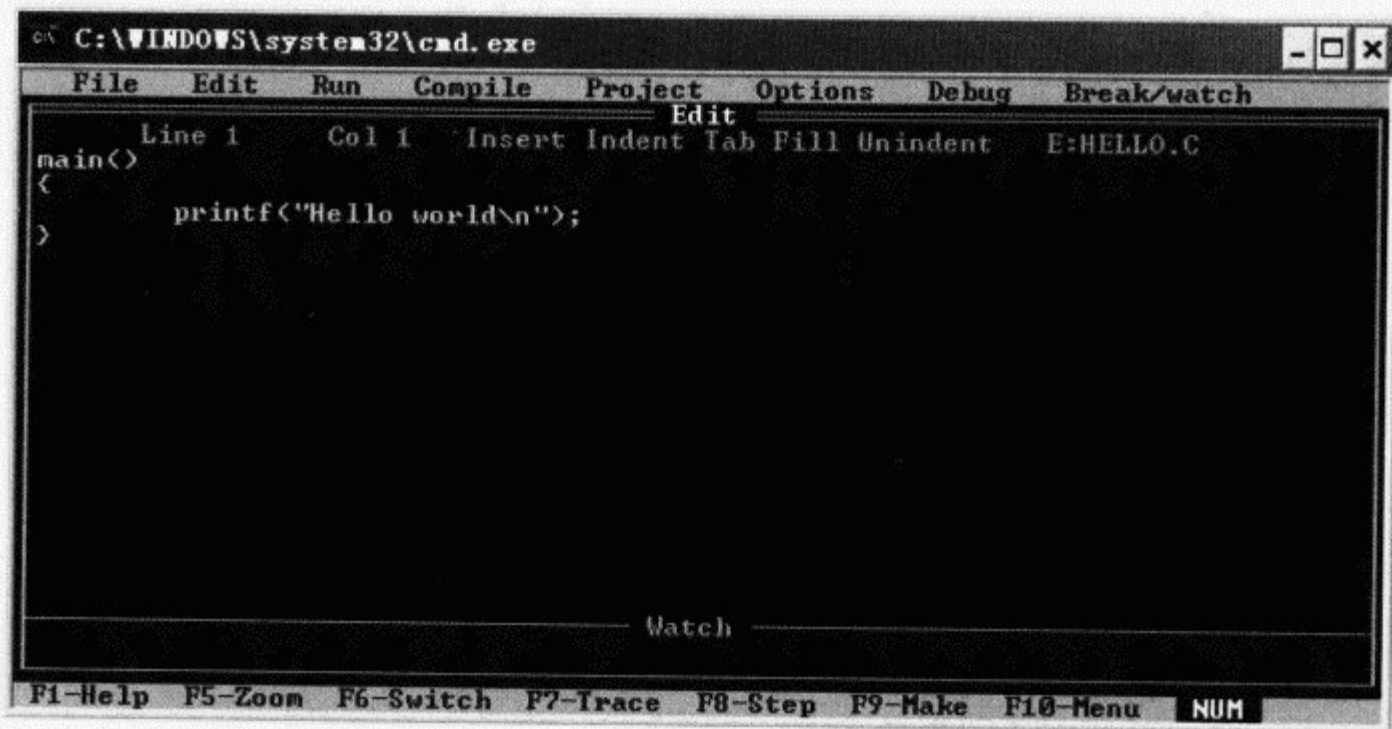


图 1-5 在编辑区编写的 C 程序

当然也可以在其他编辑器（例如记事本 notepad）中编写源程序，但要保存成后缀名为 .C 格式的文件（这里可将源文件命名为 HELLO.C）。然后将该文件存放到前面指定的文件保存和打开路径下。

我们可以通过 File→Load 命令打开已保存的源文件，如图 1-6 所示。

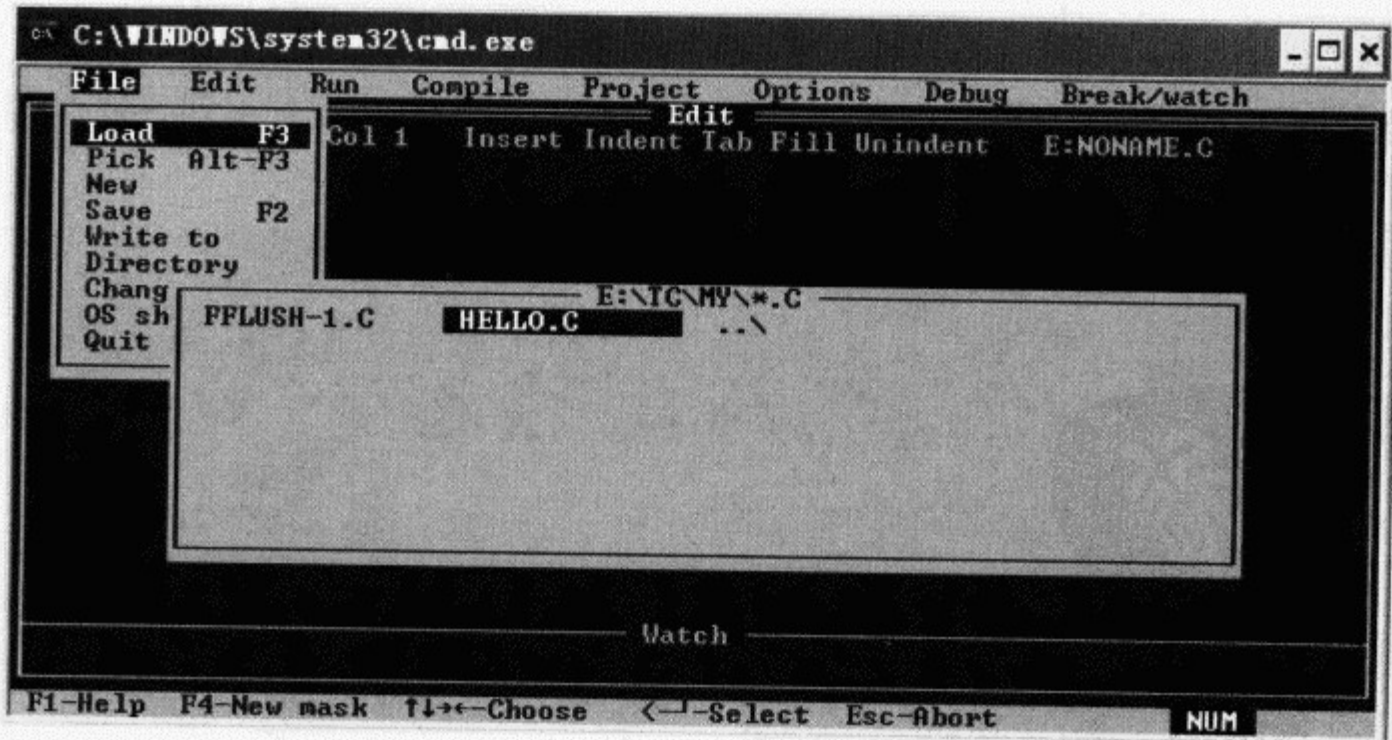


图 1-6 打开源文件

可以看到，在目录 E:\TC\MY\下已经存在编辑好的 C 文件 HELLO.C，选中这个文件名就可将该文件调入并显示在编辑区中。



### 1.3.5 源程序的编译

程序编辑好了之后,就要编译、链接源程序,最终生成可执行文件.EXE。编译源文件的方法是选择 Complier 菜单下的 Compile to OBJ 选项。具体地,按 Alt+C 组合键选中 Compile 菜单,再移动光标选中 Compile to OBJ,按 Enter 键进行编译。编译通过后会生成一个.OBJ 文件。编译后,系统会给出编译信息,在屏幕上显示出有几个错误,或是没有错误通过编译。编译成功的界面如图 1-7 所示。

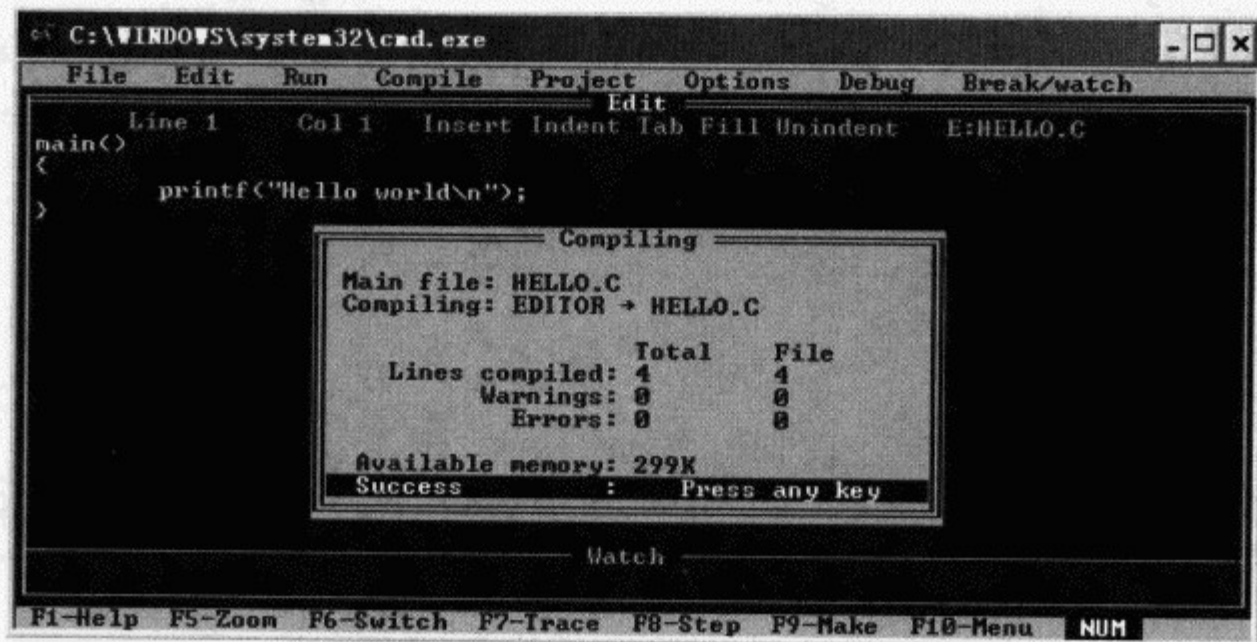


图 1-7 程序编译成功

编译通过意味着编写的源程序在语法上没有错误,这样就可以进入到链接的环节。

### 1.3.6 程序的链接

所谓程序的链接,是将编译生成的.OBJ 文件链接成.EXE 文件执行。方法是选择 Complier 菜单下的 Link EXE file,这样就会在指定的目录下生成一个.EXE 可执行文件。同样,链接后系统会给出链接信息,在屏幕上显示出有几个错误,或是没有错误通过链接。链接成功的界面如图 1-8 所示。

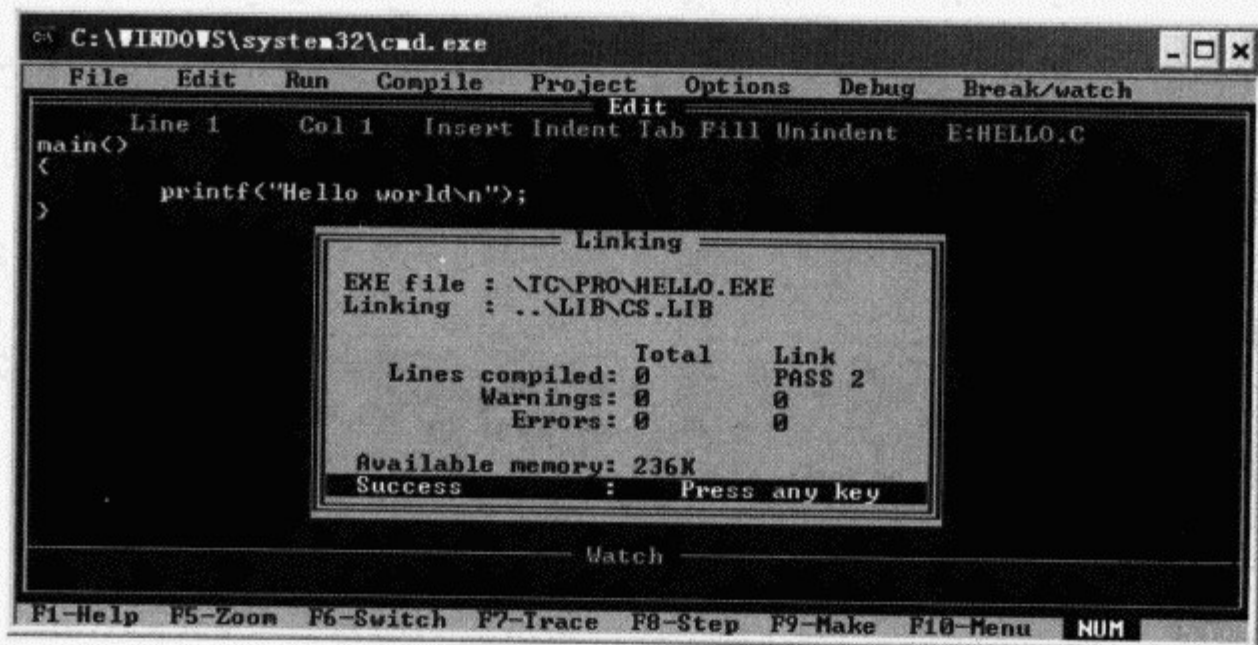


图 1-8 程序链接成功



如果程序链接成功,系统就会在指定的路径(这里设定为 E:\TC\PRO\)下生成一个.EXE 文件,这就是可执行的 C 程序。

### 1.3.7 程序的运行

最后是执行生成的.EXE 可执行文件。选择 Run 菜单下的 Run 选项,系统就会自动执行生成的.EXE 文件。按 Alt+F5 组合键就可以查看运行结果,再按任意键,返回 TC 集成开发环境窗口。程序运行的结果如图 1-9 所示。



图 1-9 程序运行的结果

当然,如果直接按 Ctrl+F9 组合键,就可以省去前面提到的编译、链接、执行三个过程,其实它是将上述三个过程合为一体。

如果编译链接程序不通过,就需要回到 Edit (编辑) 状态重新修改源文件,直到通过编译链接为止。这里需要注意的是,当编译或链接程序出错时,系统会在屏幕上提示出错信息,这时按任意键,系统就会在源文件中指示出错的位置,以便程序员修改。但有时提示出错的位置也未必准确,这就需要程序员在提示附近仔细查找。另外,这里讲的系统自动查找错误只是语法上的错误,并不涉及语义上的错误或是算法上的错误。错误为零只表明程序可以执行,但并不能保证程序执行正确。

### 1.3.8 保存源文件退出 Turbo C 环境

最后的工作应该是保存源文件并退出 Turbo C 环境。通过 File→Save 命令保存源程序。如果不改动存储路径,系统会将源程序保存在最初设置的文件保存和打开路径下,在这里就是 E:\TC\MY\目录下,如图 1-10 所示。

如图 1-10 所示,系统将文件保存在 E:\TC\MY\目录下,这里需要用户给该文件命名,以.C 作为后缀,否则系统会将文件命名为 NONAME.C。当然,存储的路径也可以改变,但需要用户自己输入。



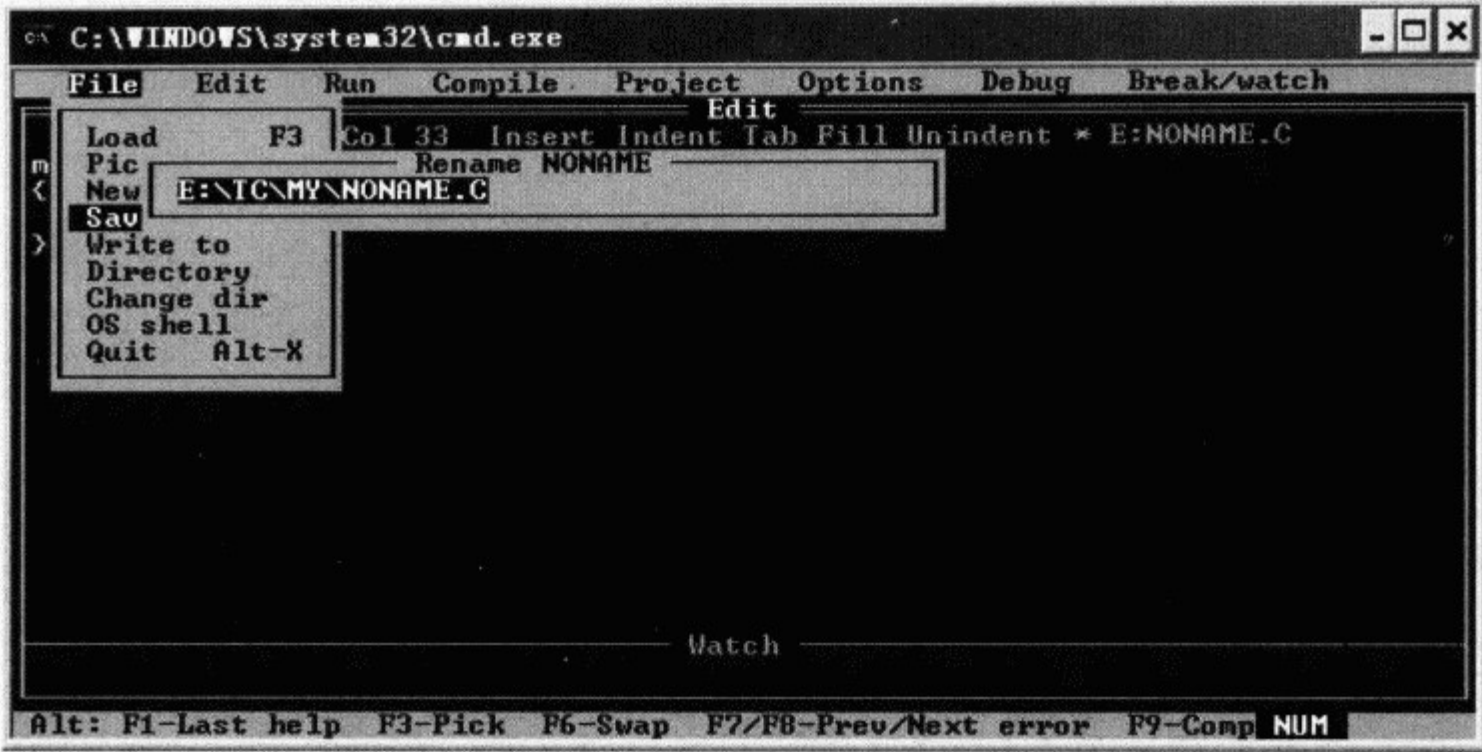


图 1-10 源文件的保存

退出 Turbo C 的方法很简单，只要按 Alt+X 组合键即可退出系统。如果文件已保存，则可直接退出系统；如果文件尚未保存，Turbo C 会给出是否保存当前文件的提示。

### 1.4 解析最简单的 C 程序

上节是从整个 C 程序开发流程的角度，以编写一个简单的“Hello world”程序为例介绍 Turbo C 环境的使用。本节将从 C 程序本身出发，以例程 1-1 所示的 C 程序为例来介绍什么是 C 程序。

**例程 1-1** 一个最简单的 C 程序。

```
#include <stdio.h>
int main(void)
{
    printf("Hello world!");
    return 0;
}
```

例程 1-1 实现的功能是在屏幕上显示“Hello world!”字符串。该段程序包括两个函数：主函数 main 和库函数 printf，在主函数 main 中调用 printf 函数。printf 函数用于向终端输出字符串。本例程的执行结果是：

```
Hello world!
```

上面这段程序是一段再简单不过的程序了，但是它包含了一个 C 程序所必需的基本要素。首先，无论多么简单或是多么复杂的 C 程序，要顺利地执行都必须有一个程序的入口，这个入口就是主函数 main()。C 程序的基本单位就是函数，一个 C 程序就是由一个主函数和若干个函数构成的。主函数调用其他函数，其他函数之间也可以相互调用。而主函数是由系统调用的。每个 C 程序必须有一个主函数，函数体部分用“{}”括起来。执行一段 C



程序,必须找到该程序的主函数,从主函数起执行,并在执行的过程中不断调用其他函数。例程 1-1 中 `int main(void)` 就是主函数的声明,等待着系统的调用。

除此之外,主函数还要调用其他函数完成程序希望实现的功能。这些函数有用户自己定义的,还有一部分是所谓的库函数。库函数并非 C 语言的一部分,它是由设计者根据编程需要自己编制并提供给用户使用的。这些库函数一般定义在头文件(.h 文件)中,因此要调用系统提供的库函数,就要在源程序开头包含该库函数所在的头文件,用 `#include<file.h>` 的形式包含头文件。例程 1-1 中调用的 `printf` 函数是在 `<stdio.h>` 头文件中定义的,因此要在源程序开头包含该头文件。有关函数以及 C 语言库函数的知识在后续章节中会做详细介绍。

## 1.5 本章小结与要点回顾

本章主要介绍了 C 语言的产生与发展、C 语言的特点及 C 程序的开发平台 Turbo C,又编写了一个简单的 C 程序,用以说明 C 程序的结构和基本编写方法。

### 1. C 语言的产生与发展大致分为六个阶段

- (1) ALGOL 60 阶段,ALGOL 60 是一种面向问题的程序设计语言,本身具有局限性,不适于编写系统程序。
- (2) 剑桥大学推出 CPL 语言,但规模较大,难以实现。
- (3) 对 CPL 语言的简化,推出 BCPL 语言。
- (4) 在 BCPL 语言的基础上,设计出既简洁又接近硬件的 B 语言。
- (5) 贝尔实验室在 1972—1973 年间,在 B 语言的基础上又设计出了 C 语言。
- (6) C 语言的后续发展,直至国际标准化组织(International Standard Organization, ISO)接受 87 ANSI C 作为 ISO C 的标准。

### 2. C 语言的特点

- (1) 语言简洁、使用方便、易学易用。
- (2) 运算符丰富。
- (3) 数据类型丰富。
- (4) 语法限制不太严格,编程自由度大。
- (5) 程序设计结构化。
- (6) 具有位操作能力,能够直接访问物理地址。
- (7) 代码质量高,程序的可移植性强。

### 3. C 程序的开发平台 Turbo C

- (1) Turbo C 的特点:集程序的编辑、编译、链接和运行于一身的集成开发环境,界面友好、直观、操作方便。
- (2) 编写一个 C 程序的基本步骤:编辑→编译→链接→执行。



#### 4. 一个简单的C程序示例

一个C程序的组成要素：主函数——程序的入口；一般函数——由主函数调用。



## 数据类型和运算符

了解了 C 语言的产生、发展，C 语言的特点以及开发环境后，从本章开始，将要深入探讨 C 语言的相关知识。本章主要介绍 C 语言中的数据类型和运算符。从第 1 章中我们了解到 C 语言的运算符和数据类型十分丰富，也正是因为 C 语言具备这样的特点，使得它的运算能力大大增强，从而可以实现其他高级语言难以实现的运算。同时，它也可以实现较复杂的数据结构，从而提高编程质量。

数据类型是程序中数据的组织形式，运算符则是对程序中数据进行运算的工具。一个程序离不开数据，而数据要在内存中妥善存储和安全传输，就需要定义它的数据类型；数据间要进行运算处理就需要运算符的帮助。因此，掌握 C 语言的数据类型和运算符是学习 C 语言的基础。

### 2.1 常量与变量

程序中的数据分为两种，一种是常量，一种是变量。

#### 1. 常量

常量就是指在程序运行过程中其值保持不变的量。在程序中，以数字形式出现的如 1、2、1.5，以字符形式出现的如 'a'、'b'，以字符串形式出现的如 "This is a test" 等都是常量。这种常量称为直接常量。

#### 例程 2-1 程序中的直接常量。

```
#include <stdio.h>
int main(void)
{
    printf("This is a test\n");
    printf("%d\n", 10);
    printf("%ld\n", 100000);
    printf("%f\n", 3.14);
    printf("%c\n", 'a');
    return 0;
}
```

在本段例程中，"This is a test"、10、100000、3.14、'a' 都是直接常量，它们不能够被改变。但是需要注意的是，它们虽然是常量，但它们的数据类型却不同，也就是说它们在内存中的存储空间大小是不一样的。有关数据类型的相关知识接下来介绍。

除了直接常量外，还可以用标识符代表一个常量。特别是一些具有特定意义的常量，



用标识符代表更直观，代码的可读性增强，同时又减少了常量的输入次数，特别是在修改常量时更加方便。

### 例程 2-2 符号常量的使用。

```
#include <stdio.h>
#define PI 3.14
int main(void)
{
    double d=10;
    double circle;
    circle=d*PI;
    printf("%lf",circle);
    getchar();
    return 0;
}
```

本例程中将符号 **PI** 定义为 3.14。这里是用 **#define** 命令定义的。程序是求圆的周长，其中将 **PI** 定义为 3.14。这样，在程序中凡是遇到 **PI** 这个符号，就代表了 3.14，这使得代码更加规范，可读性更强。此外，如果要修改 **PI** 值，也只需在定义处修改一次，程序的其他地方并不需要改动。例如，将 **PI** 改为 3.1416 只需将定义修改为：

```
#define PI 3.1416
```

## 2. 变量

变量，顾名思义就是可以改变的量。在程序设计语言中，变量的意思就是用一个变量名标记一个内存空间来存放数据。只要这个空间在内存中存在，变量名一定，那么变量中的内容是可以变化的，因此叫做变量。我们可以从 3 个方面来理解变量：变量名、变量值和变量所占内存空间。

- (1) 变量名：标记内存空间的符号地址，一个变量名对应一个内存空间和一个变量值。
- (2) 变量值：变量名对应的内存空间所存放的数据。
- (3) 变量所占内存空间：变量值实际存储的物理空间。

例如例程 2-2 中，程序中定义了变量 **double circle**；其中 **double** 是该变量的类型，用来规定变量在内存中占据空间的大小。比如：**char** 类型的数据在内存中占 1 字节，**double** 类型的数据在内存中占 8 字节等。**circle** 是变量名，它标志着一段内存空间。程序员可以向变量 **circle** 中存放任何符合 **double** 类型的数据，这个数据就叫做变量值。变量值是可变的，也就是说变量 **circle** 中可以存放不同的值。

## 3. 标识符

介绍了常量、变量的概念，接下来将引入标识符的概念。标识符 (**identifier**) 是指用来标识变量、符号常数、函数、数组、类型、文件等的有效字符序列。前面提到的 **double**、**circle**、**PI** 都可以称为标识符。简言之，标识符就是一个名字。

C 语言中规定标识符只能由字母、数字、下划线三种字符组成。而且第一个字符只能是字母或下划线。像以下列出的都是合法的标识符：

```
i, j, sum, average, tmp, c, std_num, _1, _2, Hength, Length
```



像以下列出的就是非法标识符：

```
B.W.Kernighan, 123, 3x, $, a+b
```

这里需要注意几点：

(1) C 语言是区分大小写的，因此即便是同样的字母或单词，大小写不同则代表不同的标识符。例如：i 和 I 就是两个不同的变量名。一般地，变量名都用小写字母表示。

(2) 在规定变量名时，应该做到“见名知意”，以增加程序的可读性。例如：加法运算的和最好存储在变量 sum 中，中间过程变量存储在 tmp 中等。

(3) ANSI C 没有限定标识符的长度，编程时要根据不同版本的 C 编译环境制定标识符（主要是变量名）的长度。

## 2.2 C 语言中的关键字

关键字又叫保留字，用来命名 C 程序中的语句、数据类型和变量属性。C 语言共有 32 个关键字，这使得 C 语言语法简洁、使用方便、易学易用。每个关键字都有其具体的含义，不能再作他用。因此，程序员在书写程序时就不能任意起变量名、函数名，除了上面讲到的标识符书写规则外，如果用户自定义的变量名、函数名与关键字冲突也是非法的。例如：企图定义一个整型变量 long 来存放长度值：

```
int long;
```

就是非法的。因为 long 是 C 语言中的关键字，它也是数据类型的一种。

下面把 32 个关键字汇总，并给出每个关键字的含义。

- ✧ auto: 声明自动变量，这个关键字很少用。
- ✧ double: 声明双精度变量或返回值为双精度值的函数。
- ✧ int: 声明整型变量或返回值为整型值的函数。
- ✧ struct: 声明结构体变量或函数。
- ✧ break: 跳出当前循环。
- ✧ else: 条件语句否定分支（要与 if 连用）。
- ✧ long: 声明长整型变量或返回值为长整型值函数。
- ✧ switch: 用于开关语句（另一种条件语句）。
- ✧ case: 开关语句分支。
- ✧ enum: 声明枚举类型。
- ✧ register: 声明寄存器变量。
- ✧ typedef: 自定义数据类型。
- ✧ char: 声明字符型变量或返回值为字符型值的函数。
- ✧ extern: 声明变量是在其他文件中声明（也可以看作是引用变量）。
- ✧ return: 子程序返回语句（可以带参数，也可以不带参数）。
- ✧ union: 声明联合数据类型。



- ◇ `const`: 声明只读变量。
- ◇ `float`: 声明浮点型变量或返回值为浮点型值的函数。
- ◇ `short`: 声明短整型变量或返回值为短整型值的函数。
- ◇ `unsigned`: 声明无符号类型变量或函数。
- ◇ `continue`: 结束当前循环, 开始下一轮循环。
- ◇ `for`: 循环语句。
- ◇ `signed`: 声明有符号类型变量或函数。
- ◇ `void`: 声明函数无返回值、无参数或无类型指针 (基本上就这三个作用)。
- ◇ `default`: 开关语句中的其他分支。
- ◇ `goto`: 无条件跳转语句, 结构化程序设计中不建议使用。
- ◇ `sizeof`: 计算数据类型长度。
- ◇ `volatile`: 说明变量在程序执行中可被隐含地改变。
- ◇ `do`: 循环语句的循环体。
- ◇ `while`: 循环语句的循环条件。
- ◇ `static`: 声明静态变量。
- ◇ `if`: 条件语句。

以上简要介绍了 32 个关键字的名称和含义。在后续章节中将其做详细的阐述。读者需注意的是, 关键字都有其固定的含义和用途, 要“专字专用”, 不可以任意使用, 否则就要出错。

## 2.3 C 语言的基本数据类型

程序中的数据分为常量和变量两种, 不论是常量还是变量都要有其数据类型。这就好比世界上的人, 分为男人和女人, 而每个人都要有其国籍。所谓数据类型, 就是指数据在内存中的组织形式, 它是数据的属性。比如, 整型数据在内存中存储占 2 字节, 字符型数据在内存中存储占 1 字节。

常量的数据类型是在声明时自动给出的。例如 5、6、-2 为整型常量, 2.3、3.14 为浮点型常量, 'a'、'b' 为字符型常量。而只要定义了一个变量, 就必须规定该变量的数据类型。这样, 在程序编译时, 系统会按照程序员规定的变量类型为变量分配内存空间。例如: `int a` 意思就是定义一个整型变量 `a`。在程序编译时, 系统会开辟一个 2 字节大小的内存空间, 以 `a` 为符号地址 (名字为 `a`), 程序可以向 `a` 变量中存储任何整型数。

C 语言中的数据类型归纳如下。

- (1) 基本类型: 整型、浮点型、字符型、枚举类型。
- (2) 构造类型: 数组类型、结构体类型、联合类型。
- (3) 指针类型。
- (4) 空类型。

基本类型在 C 语言编程中广泛使用, 构造类型和指针类型都要以基本类型为基础。各



类型定义的变量、常量在内存中占据的空间都不相同，在程序设计中的用法也不一样。

本章重点介绍4种基本类型。构造类型、指针类型和空类型将在后续章节中详细介绍。

### 2.3.1 整型

#### 1. 整型常量

整型常量就是整数类型的常量。在C语言中，整型常量有三种表示方法。

(1) 十进制整数：像1、2、-25、0都是十进制整数。

(2) 八进制整数：以0开头的数为八进制整数表示。如0123就表示八进制整数123，换算为十进制整数得83。

(3) 十六进制整数：以0x开头的数为十六进制整数表示。如0x123就表示十六进制整数123，换算为十进制整数得291。

#### 2. 整型变量

整型变量就是整数类型的变量。定义一个整型变量就是在内存中开辟一个空间（一般为2字节大小），以程序员的命名为标记，可以向该变量中存储整型数。这里要注意的是，数据在计算机中以二进制形式存储，因此2字节大小就是16位二进制数（0，1）。通过数据在计算机内部表示的相关知识我们知道，用不同形式的码制存储数据，数据表示的范围是不一样的。例如：原码和补码表示的数据范围就不一样。因此，同样是整型变量，由于数据存储的码制不同，可将整型变量进一步划分。整型变量可分为三类：

- (1) 基本整型 int。
- (2) 短整型 short int。
- (3) 长整型 long int。

这些整型表示都是以补码形式存储数据，也就是说，int、short int、long int 的表示范围既有正数又有负数。例如 int 类型变量的取值范围是 $-2^{15} \sim 2^{15}-1$ ，即-32 768~32 767。在实际的应用中，有一些数据是不需要符号的，为了有效地利用存储空间，更大范围地表示整数，C语言规定可以将变量定义为“无符号”类型。其实就是将存储的第一位不作为符号位，而只作为数据位的第一位。这样表示的数据一律为正，就没有负数问题了。如果加上修饰符 unsigned 就表示定义的变量为无符号类型，如果加上修饰符 signed 就表示定义的变量为有符号类型。一般情况下，如果不加 signed 修饰符，就默认为有符号类型。

通过一个例子来比较一下无符号整型变量和有符号整型变量。例如 int a 中，a 为有符号的整型变量，因此 a 的取值范围是 $-2^{15} \sim 2^{15}-1$ ，即-32 768~32 767。而 unsigned int a 中，a 为无符号的整型变量，因此 a 的取值范围是 $0 \sim 2^{16}-1$ ，即0~65 535。

这样，两个修饰符 signed、unsigned 同三类整型 int、short int、long int 就可组成以下6种整型。

- (1) 无符号基本型：unsigned int 或 unsigned。
- (2) 有符号基本型：signed int 或 int。



- (3) 无符号短整型: unsigned short int 或 unsigned short。
- (4) 有符号短整型: signed short int 或 short int。
- (5) 无符号长整型: unsigned long int 或 unsigned long。
- (6) 有符号长整型: signed long int 或 long int。

上述 6 种整型的 ANSI 标准定义如表 2-1 所示。

表 2-1 ANSI标准定义的整型

类型说明符	数的范围		字节数
int	-32 768~32 767	即 $-2^{15} \sim 2^{15}-1$	2
unsigned int	0~65535	即 $0 \sim 2^{16}-1$	2
short int	-32 768~32 767	即 $-2^{15} \sim 2^{15}-1$	2
unsigned short int	0~65 535	即 $0 \sim 2^{16}-1$	2
long int	-2 147 483 648~2 147 483 647	即 $-2^{31} \sim 2^{31}-1$	4
unsigned long	0~4 294 967 295	即 $0 \sim 2^{32}-1$	4

下面通过例子来进一步理解整数类型的概念。

例程 2-3 整型变量的定义与使用。

```
#include <stdio.h>
int main(void)
{
    int a,b;           /*定义 a,b 为整型变量*/
    int sum;           /*定义 sum 为整型变量*/
    a=2;               /*给变量 a 赋值*/
    b=-3;              /*给变量 b 赋值*/
    sum=a+b;           /*计算 a+b, 并将结果赋值给 sum*/
    printf("%d",sum);  /*在屏幕上显示计算结果, 即 sum 的值*/
    getch();
    return 0;
}
```

本例程中，首先定义 a、b 和 sum 为整型变量。这样就在内存中开辟了 3 个大小各为 2 字节的空间，并命名为 a、b、sum。然后将 2 赋值给变量 a，将-3 赋值给变量 b（赋值语句接下来讨论）。其实就是将 2 和 3 分别存储到 a、b 两变量中。再将 a 与 b 相加，并将结果赋值给变量 sum。最后通过 printf 函数将 sum 的值显示在屏幕上。从该例程中可以看出，变量的本质就是待存储数据的内存空间，而整型变量就是规定了一定大小的变量，用以存放整数。本例程的运行结果是：

-1

例程 2-4 整型数据的溢出。

```
#include <stdio.h>
int main(void)
{
    unsigned a;        /*定义 a 为无符号整型变量*/
    int b,c;           /*定义 b,c 为整型变量*/
    unsigned sum1,sum2; /*定义 sum1,sum2 为无符号整型变量*/
    a=65534;           /*给变量 a 赋值为 65534*/
    b=1;               /*给变量 b 赋值为 1*/
    c=2;               /*给变量 b 赋值为 2*/
}
```



```

sum1=a+b;          /*计算 a+b, 并将结果赋值给 sum1*/
sum2=a+c;          /*计算 a+c, 并将结果赋值给 sum2*/
printf("%u\n",sum1); /*在屏幕上显示计算结果, 即 sum1 的值*/
printf("%u\n",sum2); /*在屏幕上显示计算结果, 即 sum2 的值*/
getch();
return 0;
}

```

本例程首先定义 a、sum1、sum2 为无符号整型变量, 定义 b、c 为整型变量。然后给 a 赋值为 65534, 给 b 赋值为 1, 给 c 赋值为 2。之后分别求和并将 a 与 b 的和赋值给 sum1, 将 a 与 c 的和赋值给 sum2。屏幕上显示的运算结果为:

```

65535
0

```

第一个运算结果显然正确, 因为  $65\,534+1=65\,535$ 。而第二个运算结果显然是错误的,  $65\,534+2$  应该等于 65 536, 而结果则显示的是 0。产生错误的原因就是所谓的整型数据的溢出。

前面讲到, 无符号整型的取值范围是  $0\sim 2^{16}-1$ , 即  $0\sim 65\,535$ 。也就是说, 一个无符号整型变量最大存储数据是十进制的 65 535, 如果存储的数据超过这个上限值, 就叫做数据的溢出。在计算机内部整数的表示如图 2-1 所示。

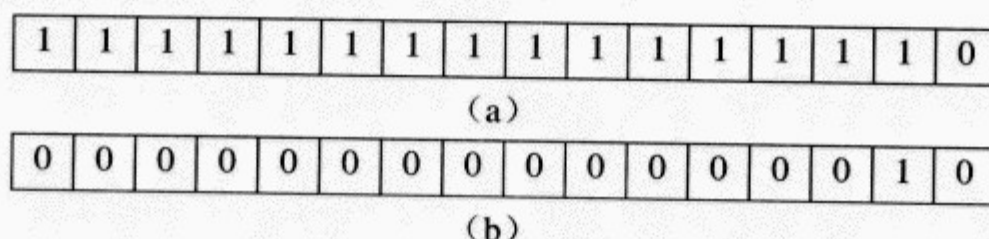


图 2-1 计算机内部整数的表示

图 2-1 中 (a) 表示整数 65 534, 其二进制表示为  $(1111111111111110)_2$ , (b) 表示整数 2, 其二进制表示为  $(0000000000000010)_2$ 。那么 (a)+(b) 用二进制加法就得  $(10000000000000000)_2$ , 共 17 位。而内存空间只有 16 位, 因此舍去最高位得  $(0000000000000000)_2$ 。于是计算便出现了错误。所以编写程序时必须注意这个问题, 要考虑计算的结果是否在变量规定的范围内。本例程只要将变量 sum1、sum2 和 a 的类型改为长整型 long, 就能容纳下计算的结果, 从而保证计算不溢出。例程 2-5 是对例程 2-4 的修正。

### 例程 2-5 对例程 2-4 的修正。

```

#include <stdio.h>
int main(void)
{
    long a;          /*定义 a 为无符号整型变量*/
    int b,c;          /*定义 b,c 为整型变量*/
    long sum1,sum2;   /*定义 sum1,sum2 为无符号整型变量*/
    a=65534;          /*给变量 a 赋值为 65534*/
    b=1;              /*给变量 b 赋值为 1*/
    c=2;              /*给变量 b 赋值为 2*/
    sum1=a+b;         /*计算 a+b, 并将结果赋值给 sum1*/
    sum2=a+c;         /*计算 a+c, 并将结果赋值给 sum2*/
    printf("%ld\n",sum1); /*在屏幕上显示计算结果, 即 sum1 的值*/
    printf("%ld\n",sum2); /*在屏幕上显示计算结果, 即 sum2 的值*/
    getch();
    return 0;
}

```



本例程的运行结果为：

```
65535
65536
```

## 2.3.2 浮点型

### 1. 浮点型常量

浮点型常量又称为实型常量，在 C 语言中有两种表示形式：

(1) 十进制小数形式。它和我们日常生活中表示小数的方法类似，由数字和小数点组成。例如：2.5、3.14、5.0 等都是浮点型常量的十进制小数形式。

(2) 指数形式。像  $5e3$  或  $5E3$  都是浮点型常量的指数形式，它代表  $5 \times 10^3$ 。

这里要注意，在用十进制小数表示浮点型常量时必须有小数点。即便是要表示整数，也要加上小数点。例如：5 的表示就是错误的，要写成 5.0。在用指数形式表示浮点型常量时，e 或 E 的前面要有数值，而且 e 或 E 后面的指数必须为整数。例如： $5e3$ 、 $12.5E6$  等都是正确的表示，而像  $e8$ 、 $5E3.5$  等都是不合法的表示。

标准 C 允许浮点数使用后缀，后缀为“f”或“F”。如  $123f$  和  $123.$  是等价的。

浮点数的指数形式也有不同的表示方法。例如浮点数 1234.5，用指数形式可表示为： $1234.5e0$ 、 $123.45e1$ 、 $12.345e2$ 、 $1.2345e3$ 、 $0.12345e4$ 、 $0.012345e5$ ……

这几种都是正确的表示方法，而只有  $1.2345e3$  称为“规范化的指数形式”。规范化的指数形式的特点是 e 前面的小数，小数点左边有且仅有一位非零数字。例如： $1.2345e3$ 、 $2.356E12$ 、 $6.08e23$  等都是规范化的指数形式。在编程时，如果一个浮点型数是按指数形式输出的，那么它一定是按规范化的指数形式输出的。例如将小数 1234.5 按指数形式输出，则它的输出形式为  $1.2345e+003$ 。

### 2. 浮点型变量

浮点型变量就是在内存中开辟一个空间(一般为 4 字节大小)，以程序员的命名为标记，可以向该变量中存储浮点数。在计算机内部，浮点数都是按照指数形式存储的。例如浮点数 3.14159 在内存中的存放，形式如图 2-2 所示。

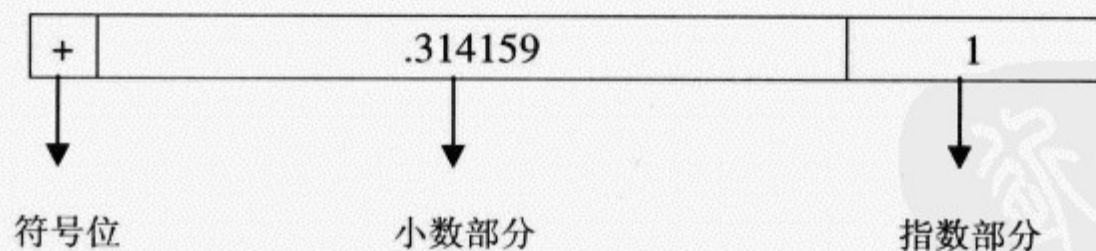


图 2-2 浮点数在内存中的存放形式

在计算机内部，浮点数是用二进制数表示的。也就是说，小数部分的形式不是用图中的十进制表示，而是用二进制表示；指数部分的形式也是用 2 的次幂来表示的，而不是用图中的 10 的次幂表示。因此，对小数部分和指数部分位数取的不同，表示的浮点数的范围也就不一样。具体地说，小数部分位数取的越多，浮点数表示越精确；指数部分位数取的



越多，能表示的浮点数范围越大。

在 C 语言中，浮点型变量分为单精度型 (float)、双精度型 (double)、长双精度型 (long double) 三种类型。在 Turbo C 中单精度型占 4 字节 (32 位) 的内存空间，其数值范围为 3.4E-38~3.4E+38，只能提供 7 位有效数字。双精度型占 8 字节 (64 位) 内存空间，其数值范围为 1.7E-308~1.7E+308，可提供 16 位有效数字。具体的浮点型数据的规定如表 2-2 所示。

表 2-2 Turbo C 中浮点型数据的规定

类型说明符	位数 (字节数)	有效数字	数的范围 (数量级)
float	32(4)	6~7	$10^{-37} \sim 10^{38}$
double	64(8)	15~16	$10^{-307} \sim 10^{308}$
long double	128(16)	18~19	$10^{-4931} \sim 10^{4932}$

下面通过例子来进一步理解浮点型的概念。

例程 2-6 浮点型变量的定义与使用。

```
#include <stdio.h>
#define PI 3.14                /*定义浮点型常量*/
int main(void)
{
    float d;                   /*定义浮点型变量 d*/
    float circle;              /*定义浮点型变量 circle*/
    d=10.0;                    /*将浮点数 10.0 赋值给变量 d*/
    circle=d*PI;               /*将 d*PI 的结果赋值给 circle*/
    printf("%f\n",circle);     /*以十进制小数形式输出结果*/
    printf("%e\n",circle);     /*以指数形式输出结果*/
    getchar();
    return 0;
}
```

本例程首先定义浮点型常量 PI，即语句 #define PI 3.14，然后定义浮点型变量 d，circle，并将 10.0 赋值给变量 d。再将 d\*PI 的积赋值给变量 circle。最后，在屏幕上以两种方式显示计算结果，即 circle 的值。一种是用 “%f” 输出一个浮点型的十进制小数形式，一种是用 “%e” 输出浮点型的指数形式。本例程的运行结果是：

```
31.400000
3.140000e+01
```

从该例中可以看到，浮点型变量的定义与使用同整型变量类似。输出的形式也可以根据用户需要而定，可以以十进制小数形式和指数形式输出结果。

例程 2-7 浮点数的舍入误差。

```
#include <stdio.h>
int main(void)
{
    float a;                   /*定义单精度浮点型变量 a*/
    double b;                  /*定义双精度浮点型变量 b*/
    a=33333.33333;             /*将浮点数 33333.33333 赋值给 a */
    b=33333.3333333333333333;  /*将浮点数 33333.3333333333333333 赋值给 b */
    printf("%f\n%f\n",a,b);    /*输出结果*/
    getchar();
}
```



```
    return 0;  
}
```

本例程的运行结果为：

```
33333.332031  
33333.333333
```

在本例程中，将浮点数 33333.33333 赋值给单精度浮点型变量 a，将浮点数 33333.3333333333333333333333333333 赋值给双精度浮点型变量 b。由于 a 是单精度浮点型，有效位数只有 7 位。而整数已占 5 位，故小数点两位后均为无效数字。因此，单精度浮点数的输出只有 7 位是正确的，后 4 位为无效数字。双精度浮点有效位为 16 位，故后 3 位为无效数字。但是编译系统 Turbo C 规定小数后最多保留 6 位，其余部分四舍五入。因此双精度的浮点数 b 显示为 33333.333333，后面的位四舍五入了。因此也就产生了所谓的浮点数的舍入误差。也就是超出了浮点数的有效数位数。

由于浮点数存在有效位数和舍入误差的问题，所以在编程时注意不要超出浮点数的有效数位数。例如不要将一个很大的浮点数与一个很小的浮点数相加或相减，否则会失去相加的意义因为很大的浮点数后面几位可能是无效数字。

## 2.3.3 字符型

### 1. 字符型常量

在程序中，用单引号括起来的字符称为字符型常量。例如：'a'，'3'，'='，'\*'，'?'等都是合法字符常量。在 C 语言中，字符型常量是区分大小写的。例如字符'a'和字符'A'是不同的字符型常量。

在 C 语言中，字符常量有以下特点：

- (1) 字符常量只能用单引号括起来，不能用双引号或其他括号，双引号括起的是字符串。
- (2) 字符常量只能是单个字符，不能是几个字符。
- (3) 字符可以是字符集中任意字符。数字被定义为字符型之后就不能参与数值运算。如'5'和 5 是不同的。'5'是字符常量，不能参与运算。

在 C 语言中，还有一类特殊的字符常量称为转义字符。该类字符常量是一个以“\”（反斜杠）开头的字符序列。其实，这种转义字符在前面已多次遇到，例如 printf 函数中的'\n'，'\t'等。特别是字符'\n'，会经常出现在编程中。这些字符有别于其他字符，它们是控制字符。这些字符在屏幕上是无法显示的，而且在程序中也无法用一般形式表示。因此我们说转义字符是用来表示那些用一般字符不便于表示的控制代码。

C 语言中转义字符意义及相应的 ASCII 代码如表 2-3 所示。

表 2-3 中的最后两行是通用转义字符，应用它们可以表示 C 语言字符集中的任何一个字符。其中'\ddd'表示 1~3 位八进制的 ASCII 代码，'\xhh'表示 1~2 位十六进制的 ASCII 代码。例如：'\101'表示字母'A'，'\102'表示字母'B'，'\XOA'表示换行等。



表 2-3 C语言中常用的转义字符

转义字符	意义	ASCII代码
\n	换行	10
\t	水平制表	9
\b	退格	8
\r	回车	13
\f	换页	12
\\	反斜杠	92
\'	单引号字符	39
\"	双引号字符	34
\a	响铃	7
\ddd	1~3位八进制数所代表的字符	
\xhh	1~2位十六进制数所代表的字符	

2. 字符型变量

字符型变量是在内存中开辟一个空间（一般为 1 字节大小），以程序员命名为标记，可以向该变量中存储字符。在存储上，字符值是以 ASCII 码的形式存放在变量的内存单元之中的。例如：字符'a'的 ASCII 码为 97，存储在内存单元中就是将 97 存入。当然在计算机内部，这个数值使用二进制数表示，存储的空间大小为 1 字节，即 8 位。如图 2-3 所示为字符'b'的计算机内部表示。

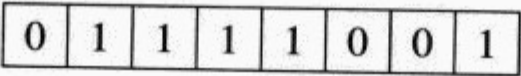


图 2-3 字符'b'的计算机内部表示

下面通过例子来进一步理解字符型常量、变量的概念。

例程 2-8 字符型变量的定义和转义字符。

```
#include <stdio.h>
int main(void)
{
    char a,b;
    a='A';
    b='\101';
    printf("%c,%c\n",a,b);
    getchar();
    return 0;
}
```

本例程中，先定义两个字符型变量 a、b，再将字符'A'赋值给变量 a，将转义字符'\101'赋值给变量 b。其中转义字符'\101'表示字母'A'。实际上两个字符型变量 a、b 存储的都是字符'A'的 ASCII 值。本例程的运行结果为：

A,A

例程 2-9 字符型变量的内部表示。

```
#include <stdio.h>
int main(void)
```



```

{
    char a,b;           /*两个字符型变量 a、b*/
    a=65;               /*将 65 赋值给变量 a*/
    b=66;               /*将 66 赋值给变量 b */
    printf("%c,%c\n",a,b); /*以字符形式显示结果*/
    printf("%d,%d\n",a,b); /*以整数形式显示结果*/
    getchar();
    return 0;
}

```

本例程中，定义两个字符型变量 a、b，但将 65 赋值给变量 a，将 66 赋值给变量 b。实际上字符型变量的内部存储形式就是存储字符的 ASCII 代码，上述的赋值方法就是将 ASCII 码直接赋值给字符型变量，因此这种赋值方法是允许的。当以字符形式显示时，即“%c”，就显示字符型变量中的 ASCII 码对应的字符；当以整数形式显示时，即“%d”，就显示字符型变量中存储的整数。本例程的运行结果为：

```

A,B
65,66

```

注意：字符型变量的存储空间是 1 字节，因此它只能存放 0~255 的整数。如果以整数形式赋值字符型变量，超过 255 就会溢出。

### 2.3.4 枚举类型

枚举类型是 ANSI C 新标准增加的数据类型。所谓枚举类型，就是可以将变量所有可能取得的值一一列出的类型。在现实生活中，这样的例子很多。例如：“工作日”可看作一个枚举类型，因为工作日就是星期一到星期五。因此，如果一个变量只有几种可能的取值，就可以定义为枚举类型。

同前面所述的类型一样，枚举类型也要声明相应的变量。不同的是，在声明枚举类型变量之前先要定义这个类型。以“工作日”为例，定义一个枚举类型如下。

```
enum workday {mon,tue,wed,thu,fri};
```

以上就定义了一个“工作日”的枚举类型。定义枚举类型要用关键字 **enum** 开头，然后是自定义的类名，大括号里的叫做枚举元素，或叫做枚举常量。枚举类型定义的一般形式如下：

```
enum 枚举名{枚举元素表};
```

定义了一个枚举类型 **enum workday** 之后，就可以定义枚举变量了。定义方法如下：

```
enum workday a,b;
```

a, b 就是定义好的枚举变量，它们的值只能从 mon, tue, wed, thu, fri 中选取其一。例如：

```

a=mon;
b= tue;

```

还有一种定义方法，就是直接定义枚举变量，如下：

```
enum workday {mon,tue,wed,thu,fri}a,b;
```



即在定义枚举类型时直接定义枚举变量。

枚举类型在使用中有以下规定：

(1) 枚举元素是常量而不是变量，因此枚举元素也叫做枚举常量，不能在程序中用赋值语句对它赋值。例如对枚举 `workday` 的元素再作以下赋值：

```
mon=1;
tue=2;
```

都是错误的。

(2) 枚举元素本身由系统定义了一个表示序号的数值，从 0 开始顺序定义为 0, 1, 2……。如在 `workday` 中，`mon` 值为 0, ..., `fri` 值为 4。

(3) 也可以改变枚举元素的值，定义时由程序员给定。例如：

```
enum workday {mon=1,tue,wed,thu,fri};
```

定义 `mon` 为 1，以后的顺序加 1，`fri` 为 5。这样与实际情况相符。

(4) 枚举值可以用来比较大小。例如：

```
tue>mon
```

(5) 枚举变量不能直接被赋值。例如对枚举变量赋值：

```
a=1;
b=2;
```

是错误的。因为枚举变量只能从枚举元素中取值。因此，变量 `a`、`b` 也只能从 `mon`, `tue`, `wed`, `thu`, `fri` 中取值。或者做一步强制转换，将整数强制转换为枚举类型。

```
a=(enum workday)1;
b=(enum workday)2;
```

下面通过例子来理解枚举类型的用法。

### 例程 2-10 枚举类型的声明和变量的定义。

```
#include <stdio.h>
int main(void)
{
    enum workday
    { mon=1,tue,wed,thu,fri } a,b,c,d,e; /*声明枚举类型，并定义枚举变量*/
    a=mon; /*给枚举变量赋值*/
    b=tue;
    c=wed;
    d=thu;
    e=fri;
    /*显示枚举值*/

    printf("Monday:%d,Tuesday:%d,Wednesday:%d,Thursday:%d,Friday:%d",a,b,c,d,e);
    getchar();
    return 0;
}
```

本例程首先定义了枚举类型，并定义枚举变量。这里用的是在定义枚举类型的同时直接定义枚举变量，而且改变了枚举元素的值，使其与实际情况相符。然后给枚举变量赋值。最后，在屏幕上显示枚举变量的值。本例程的运行结果为：

Monday:1,Tuesday:2,Wednesday:3,Thursday:4,Friday:5

## 2.4 运算符

C语言中的运算符是很丰富的，也正因此，才使得C语言的表达式种类丰富，从而使运算功能大大增强，实现其他高级语言难以实现的运算。C语言的运算符总共有34种，在第1章中已经作了简要介绍。本节将对C语言中的运算符做进一步的讨论。

### 2.4.1 算术运算符

#### 1. C语言中基本的算术运算符主要包括以下5种

(1) 加法运算符或正值运算符“+”：作为加法运算符时为双目运算符，也就是要有两个量参与加法运算。如  $a+b$ ， $1+2$  等。作为正值运算符表示正号，为单目运算，具有左结合性，如  $+3$  等。

(2) 减法运算符或负值运算符“-”：作为减法运算符时为双目运算符，也就是要有两个量参与减法运算。如  $a-b$ ， $5-3$  等。作为负值运算符表示负号，为单目运算，具有左结合性。如  $-x$ ， $-5$  等。

(3) 乘法运算符“\*”：双目运算，具有左结合性。

(4) 除法运算符“/”：双目运算，具有左结合性。参与运算量均为整型时，结果也为整型，舍去小数。如果运算量中有一个是浮点型，则结果为双精度浮点型。

(5) 求余运算符（模运算符）“%”：双目运算，具有左结合性。要求参与运算的量均为整型。求余运算的结果等于两数相除后的余数。例如： $5\%3=2$ 。

注意：以上4种运算符（+，-，\*，/）在计算时如果参与的运算量均为整型，则结果也为整数，舍去小数。如果参与运算量中有浮点数，则计算结果为 double 型。

#### 2. 算术表达式

将算术运算符、括号和参与运算量组合起来的式子称为算术表达式。这里，参与运算量包括常量、变量、函数等。以下给出了几个合法的C算术表达式：

```
a+b
1+2
(a+1)*b
sin(x)+sin(y)
(++i)-(j++)+(k--)
```

在进行算术表达式的计算时，是有优先级的。在平时的算术运算中，有一套完整的运算符优先级规则，在C语言中，算术表达式运算同样存在规则。在表达式求值时，按运算符的优先级别执行。例如：先乘除后加减，先括号里后括号外等。如果在一个运算量两侧的运算符优先级相同时，则按运算符的结合性所规定的结合方向处理，例如： $a+b-c$ 。

所谓运算符的结合性是指同级运算时，算术运算符的结合方向。C语言中各运算符的结合性分为两种，一是左结合性（自左至右），二是右结合性（自右至左）。一般地，算术



运算符的结合性是自左至右的。也就是说，同级的运算符参与表达式的计算时自左向右。例如：算术表达式  $a+b-c+d$  计算时，先算  $a+b$ ，再将  $a+b$  的结果减  $c$ ，再将  $a+b-c$  的结果加  $d$ ，最终得到计算结果。而像赋值运算符的结合性就是自右至左的。例如  $x=y=z$ ，就应该先执行  $y=z$  再执行  $x=(y=z)$  运算。C 语言运算符中也有许多为右结合性的，在编写程序时应该特别注意。

### 3. 自增自减运算符

自增自减运算符虽然不是基本的算术运算符，但它们也是用来进行算术运算的，因此放在这里介绍。自增运算符（++）的作用就是使变量值增加 1；自减运算符（--）的作用是使变量值减 1。但是运算符在变量前后的位置不同，所表达的意思是不一样的。

- ✧ ++i: i 自增 1 后再参与其他运算。
- ✧ --i: i 自减 1 后再参与其他运算。
- ✧ i++: i 参与运算后，i 的值再自增 1。
- ✧ i--: i 参与运算后，i 的值再自减 1。

下面通过一个例子来进一步理解自增、自减运算符。

#### 例程 2-11 自增、自减运算符的应用。

```
#include <stdio.h>
int main(void)
{
    int i=8;
    printf("%d\n", ++i);
    printf("%d\n", --i);
    printf("%d\n", i++);
    printf("%d\n", i--);
    getchar();
    return 0;
}
```

本例程中，先定义整型变量  $i$  并赋初值 8，然后进行一系列自增自减运算。具体过程如下：

首先进行  $++i$  运算， $i$  自增 1 后再参与其他运算。因此这一步的运行结果为 9。

第二步进行  $--i$  运算， $i$  自减 1 后再参与其他运算。由于  $i$  此时变量值为 9，因此这一步的运行结果为 8。

第三步进行  $i++$  运算， $i$  参与运算后， $i$  的值再自增 1。由于  $i$  此时变量值为 8，因此这一步的运行结果为 8。然后  $i$  的值自行加 1。

最后一步进行  $i--$  运算， $i$  参与运算后， $i$  的值再自减 1。由于  $i$  此时变量值为 9，因此这一步的运行结果为 9。然后  $i$  的值自行减 1，最终变量  $i$  的值为 8。

本例程的运行结果为：

```
9
8
8
9
```

#### 4. 强制类型转换运算符

在讲枚举类型时已经提到，可将整数强制转换为枚举类型。

```
a=(enum workday)1;  
b=(enum workday)2;
```

其中 1、2 为整型数，只有将其转换为枚举类型才能赋值给枚举变量 a 和 b。因此可以看出，强制类型转换是通过类型转换运算来实现的。

其一般形式为：

```
(类型说明符) (表达式)  
(类型说明符) x
```

其功能是把表达式的运算结果强制转换成类型说明符所表示的类型，将变量 x 转换为类型说明符所表示的类型。例如：

```
(float) a      把 a 转换为浮点型  
(int) (x+y)    把 x+y 的结果转换为整型
```

在使用强制转换时应注意以下问题：

(1) 类型说明符和表达式都必须加括号（单个变量可以不加括号），如把(int)(x+y)写成(int)x+y，则成了把 x 转换成 int 型之后再与 y 相加了。

(2) 强制转换只是改变本次计算的临时结果，产生一个要求类型的中间变量，原来变量定义的类型未发生改变。例如：

```
double x;  
int y;  
y=(int)x;
```

x 的类型仍然是双精度类型，只是在(int)x 运算时产生了临时的整型中间变量，并赋值给变量 y。

### 2.4.2 关系运算符

#### 1. 关系运算符

关系运算又叫做比较运算，是逻辑运算的一种简单形式。和算术运算一样，关系运算也有运算结果。要实现关系运算，就必须有所谓的关系表达式，它是由比较值和关系运算符组合而成的。像“>”就是关系运算符，而“a>b”就是关系表达式。如果实际的变量 a，b 的关系真的是 a>b，例如 a 的变量值为 5，b 的变量值为 3，则关系运算 a>b 的运算结果为“真”，否则运算结果为“假”。这就是关系运算的运算结果。

C 语言中提供了 6 种关系运算符。

- ◇ <: 小于。
- ◇ <=: 小于或等于。
- ◇ >: 大于。
- ◇ >=: 大于或等于。
- ◇ ==: 等于。



✧ !=: 不等于。

应用这 6 种关系运算符，可以组成不同的关系表达式来进行关系运算。

关系运算符也具有优先次序。具体地，<，<=，>，>=的优先级相同，高于==和!=，而==和!=的优先级相同。

关系运算符都是双目运算符，均为左结合。关系运算符的优先级低于算术运算符，高于赋值运算符。例如：

a>b+c	等价于 a>(b+c)
a>b==c	等价于 (a>b)==c
a=b>c	等价于 a=(b>c)

2. 关系表达式

前面已经介绍过，要实现关系运算，就必须有所谓的关系表达式。用关系运算符将两个比较的值或表达式连接起来的式子称为关系表达式。比较的值既可以是常量如 1，2，也可以是变量如 a，b；而表达式可以是算术表达式如 1+2，关系表达式如 a>b，逻辑表达式如 a&&b，赋值表达式如 a=2 等。

关系表达式的一般形式为：

表达式 关系运算符 表达式

例如下面给出的都是合法的关系表达式。

```
a+b>c-d
x>3/2
a>(b>c)
a!=(c==d)
```

关系表达式的运算结果，叫做关系表达式的值，它是一个逻辑值，只有“真”和“假”。因为在 C 语言中没有逻辑型数据，因此用 1 代表逻辑“真”，0 代表逻辑“假”。例如，关系表达式 2==3 的值为 0，5>3 的值为 1，(a=3)>(b=5)的值为 0。

关系表达式在编程时常用在条件语句中作为分支条件，或用在循环语句中作为循环的执行条件等。

2.4.3 逻辑运算符

1. 逻辑运算符

逻辑运算符是用来将若干关系表达式或逻辑值连接起来，组成更为复杂的表达式，这个表达式称为逻辑表达式。在 C 语言中提供了 3 种逻辑运算符。

- ✧ &&: 与运算。
- ✧ ||: 或运算。
- ✧ !: 非运算。

其中，与运算（&&）和或运算（||）都是双目运算，即必须有两个逻辑运算量。非运算（!）是单目运算，它只能有一个逻辑运算量。

逻辑运算也是有结果的，其结果叫做“真值”。同关系运算类似，真值只能取两个

值：“真”或“假”。同样用 1 代表逻辑“真”，0 代表逻辑“假”。真值的取值不仅取决于参加运算的运算量或表达式的值，还取决于逻辑运算符本身。逻辑运算的真值如表 2-4 所示。

表 2-4 逻辑运算真值表

a	b	!a	!b	a&&b	a  b
0	0	1	1	0	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	1

以上真值表中，1 代表逻辑“真”，0 代表逻辑“假”。通过这个真值表，就可以求出逻辑表达式的真值。例如：(2>1)&&(1<9)的真值为 1，(2<8)&&(10<1)的真值为 0，(1<5)|| (2>0)的真值为 1，若 a 为非 0 值，!a 的真值为 0。

逻辑运算符的优先次序如下：

!(非)>&&(与)>|| (或)

其中非运算符!的优先次序最高。

而将赋值运算、逻辑运算、关系运算、算术运算放到一起时，它们的优先次序是：

赋值运算符<&&和 ||<关系运算符<算术运算符<!(非)

赋值运算的优先级最小，非运算的优先级最高。

2. 逻辑表达式

前面已经介绍过，逻辑表达式就是将逻辑运算符、若干关系表达式或逻辑值连接起来组成的表达式。逻辑表达式的值叫做“真值”，它是一个逻辑量“真”或“假”。在 C 语言中 1 代表逻辑“真”，0 代表逻辑“假”。在判断一个量的逻辑值时，以 0 代表逻辑“假”，以非 0 代表逻辑“真”。

逻辑表达式的一般形式为：

表达式 逻辑运算符 表达式

其中，表达式可以又是逻辑表达式，从而组成了嵌套的情形。  
下面通过例子来理解逻辑表达式的用法。

例程 2-12 逻辑表达式的应用。

```
#include <stdio.h>
int main(void)
{
    int score;                /*定义一个变量 score 存放分数*/
    scanf("%d",&score);      /*输入分数*/
    if(score<60)              /*判断分数段*/
        printf("Fail!\n");
    else if(score>=60&&score<85)
        printf("Qualified!\n");
    else printf("Excel!\n");
}
```



```
    getchar();  
    return 0;  
}
```

本例程用来简单地划分学生分数段。首先定义一个变量 `score` 用来存放分数。然后用 `scanf` 函数输入一个整型值作为分数。接下来再用 `if-else` 语句对分数进行划分。

`if-else` 语句的规则是：

```
if (逻辑表达式) 表达式1 else 表达式2;
```

如果逻辑表达式的真值为 1，则执行表达式 1；如果逻辑表达式的真值为 0，则执行表达式 2。

本例程的执行结果要视输入的分数而定。如果输入的分数小于 60，则表达式 `score<60` 的真值为 1，于是执行语句 `printf("Fail!\n")`。否则，如果输入的分数在 60（包括 60）到 85（不包括 85）之间，则表达式 `score>=60&&score<85` 的真值为 1，于是执行语句 `printf("Qualified!\n")`。否则执行语句 `printf("Excel!\n")`。

如果三次分别输入分数 50、75、90，本例程的执行结果为：

```
50  
Fail!  
75  
Qualified!  
90  
Excel!
```

#### 2.4.4 条件运算符

条件运算符用来处理简单的条件运算。在进行选择结构的程序设计时，一般应用 `if-else` 语句或 `switch` 语句进行条件判断（后续章节会讲到）。但有时一些简单的条件判断可以使用条件运算符来处理。

例如语句：

```
if(a>b)max=a;  
else max=b;
```

就可以用以下的条件运算符处理。

```
max=(a>b)?a:b
```

可以这样来理解这句话，如果关系表达式 `a>b` 真值为 1，则将 `a` 赋值给 `max`，否则将 `b` 赋值给 `max`，`(a>b)?a:b` 称为条件表达式。条件表达式在一定条件下可以代替 `if-else` 语句进行条件判断。

条件运算符有三个操作对象，故称为三目运算符。这也是 C 语言中的唯一一个三目运算符。由条件运算符组成条件表达式的一般形式为：

```
表达式1? 表达式2: 表达式3
```

条件表达式的求值规则为：如果表达式 1 的值为真，即真值为 1，则以表达式 2 的值作为条件表达式的值，否则以表达式 3 的值作为整个条件表达式的值。

使用条件表达式时，还应注意以下几点：

(1) 条件运算符?的运算优先级低于关系运算符和算术运算符,但高于赋值运算符。因此  $\text{max}=(a>b)?a:b$  可以去掉括号而写为  $\text{max}=a>b?a:b$ 。

(2) 条件运算符?和:是一对运算符,不能分开单独使用。

(3) 条件运算符的结合方向是自右至左。

例如:

$a>b?a:c>d?c:d$

应理解为

$a>b?a:(c>d?c:d)$

这也就是所谓的条件表达式嵌套的情形,即其中的表达式3是一个条件表达式。

应用条件运算符处理条件判断赋值问题,可以简化代码,提高程序的执行效率。因此,建议在编写程序时积极使用条件运算符。

## 2.4.5 赋值运算符

相信读者对赋值运算符早已不陌生了,它在我们编写的每一段程序中几乎都要用到。赋值运算符“=”的功能是将一个值传送到一个变量中去。由赋值运算符和变量、数值或其他表达式组成的式子叫做赋值表达式。赋值表达式的一般形式为:

变量=表达式  
变量=数值

例如以下列出的都是合法的赋值表达式。

```
x=a+b;  
i=i+1;  
a=2;
```

有关赋值表达式和赋值语句的知识在第3章中还会有详细的介绍。

## 2.4.6 逗号运算符

逗号在C语言中的作用主要有两种:一是作为分隔符使用;二是作为语句中的运算符使用。在定义同一类型的多个变量时,一种方法是多次定义:

```
int x;  
int y;  
int z;
```

这种定义变量的方法比较麻烦,增加了代码量。还有一种方法就是应用逗号作为变量之间的分隔符:

```
int x,y,z;
```

这样定义简洁、直观,代码的可读性强。

逗号作为分隔符还有一种用法,就是应用到函数中,作为参数之间的分隔符使用。

例如:

```
printf("%d,%d,%d\n",a,b,c);  
fuc(x,y,z);
```



除此之外,逗号还可以作为运算符使用。逗号作为运算符使用的情况一般有两种。

(1) 应用在 for 循环语句中。

for 循环语句的一般形式为:

```
for(表达式1; 表达式2; 表达式3){循环体语句}
```

它的执行过程是:

- 1) 先求解表达式 1。
- 2) 求解表达式 2, 若其值为真 (非 0), 则执行 for 语句中指定的循环体语句, 然后执行下面第 3) 步; 若其值为假 (0), 则结束循环, 转到第 5) 步。
- 3) 求解表达式 3。
- 4) 转回上面第 2) 步继续执行。
- 5) 循环结束, 执行 for 语句下面的一个语句。

以上简单地介绍了 for 循环语句的执行过程, 有关它的深入讲解, 将在后续章节中给出。

在 for 循环语句中, 表达式 1 和表达式 3 都可以用多个表达式组成, 这些表达式要用逗号 “,” 隔开。例如下面的循环语句:

```
for(plot=1,i=1;i<1000;i++){  
    plot=plot*i;  
}
```

其中, 表达式 1 是由两个赋值表达式 `plot=1, i=1` 组成的。这段循环语句的功能是计算 1, 2, ..., 1000 的乘积。

(2) 将若干个表达式用逗号 “,” 连接, 组成一个逗号表达式。

其一般形式为:

```
表达式1, 表达式2, ..., 表达式n
```

逗号表达式又称为“顺序求值表达式”, 它的求解过程是: 先求解表达式 1, 再求解表达式 2, ……, 最后求解表达式 n。整个逗号表达式的运算结果就是表达式 n 的值。

通过下面的例子来理解逗号表达式。

### 例程 2-13 逗号表达式的应用 (1)。

```
#include <stdio.h>  
int main(void)  
{  
    int a=2,x;  
    x=(a=a*3,a+1); /*逗号表达式*/  
    printf("%d\n",x);  
    getchar();  
    return 0;  
}
```

本例程中 a 的原值为 2, 执行逗号表达式时, 先求解 `a=a*3`, a 的值变为 6; 再求解 `a+1`, 最终逗号表达式的运算结果就是表达式 `a+1` 的值。这样就是将 7 赋值给变量 x。本例程的运行结果是:

7

**例程 2-14 逗号表达式的应用 (2)。**

```
#include <stdio.h>
int main(void)
{
    int i,j,t;
    i=2;
    j=6;
    t=i,i=j,j=t;          /*逗号表达式*/
    printf("i=%d,j=%d",i,j);
    getchar();
    return 0;
}
```

在本例程中，应用逗号表达式将变量 *i* 和 *j* 的值对调。先将 *i* 的值赋值给变量 *t*，再将 *j* 的值赋值给变量 *i*，最后将 *t* 的值赋值给变量 *j*，从而实现变量值的交换。这样，就将以前要三条语句才能完成的工作改为一条语句就能完成。本例程的运行结果是：

```
i=6,j=2
```

## 2.4.7 求字节数运算符

在程序设计中，有时需要了解一个变量或者一个类型占用内存空间的大小。像一些 I/O 函数和内存分配函数，就必须知道传输数据的大小和分配内存空间的多少。因此，就要用到求字节数运算符。

求字节数运算符为 `sizeof`。`sizeof` 有两种用法。

### (1) `sizeof` 表达式

它的运算结果是得到表达式计算结果在内存中所占据的字节数。例如：

```
a=sizeof(x=2);
```

设 *x* 为整型变量，它在内存中占据 2 字节。则 *a* 的结果值为 2。

### (2) `sizeof` (类型)

它的运算结果是得到括号中的类型在内存中所占据的字节数。例如：

```
a=sizeof(int);
```

就是计算整型的量在内存中所占据的字节数。整型的量在内存中所占据的字节数为 2，所以 *a* 的结果值为 2。

## 2.5 本章小结与要点回顾

本章主要介绍了 C 语言中的数据类型和运算符。在 C 语言中，数据类型是程序中数据的组织形式，运算符是对程序中数据进行运算的工具。学好数据类型和运算符是掌握 C 语言的基础，也是编好 C 程序的关键。下面将本章所介绍的重点知识归纳如下。



### 1. C 语言中的常量和变量

常量就是指在程序运行过程中其值保持不变的量。常量分为两种：直接常量和符号常量。

变量就是可以改变的量。我们可以从三个方面来理解变量：变量名、变量值、变量所占内存空间。

### 2. C 语言中的关键字

关键字又叫保留字，C 语言共有 32 个关键字。特别强调的是：所有的关键字都有其固定的含义和用途，都要“专字专用”，不可以任意使用，否则就会出错。

### 3. C 语言中的数据类型

C 语言中的数据类型可归纳如下。

- ✧ 基本类型：整型、字符型、浮点型、枚举类型。
- ✧ 构造类型：数组类型、结构体类型、联合类型。
- ✧ 指针类型。
- ✧ 空类型。

数据类型是数据的属性，无论常量还是变量都有其自己的数据类型。而且每一种类型定义的变量、常量在内存中的占据空间都不相同，在程序设计中的用法也不一样。

### 4. C 语言中的基本数据类型

本章详细介绍了 4 种基本数据类型。

#### (1) 整型

包括整型常量和整型变量。根据整型数据计算机内部编码的差异，将整型分为 6 种。

- ✧ 无符号基本型：unsigned int 或 unsigned。
- ✧ 有符号基本型：signed int 或 int。
- ✧ 无符号短整型：unsigned short int 或 unsigned short。
- ✧ 有符号短整型：signed short int 或 short int。
- ✧ 无符号长整型：unsigned long int 或 unsigned long。
- ✧ 有符号长整型：signed long int 或 long int。

不同种类的整型可表示数的范围不同，在内存中所占的字节数也不一样。

#### (2) 浮点型

浮点型又叫做实型，包括浮点型常量和浮点型变量。浮点型常量有两种表示方法：十进制小数形式和指数形式。

在计算机内部，浮点数都是按照指数形式存储的。根据浮点型数据在计算机内部表示的精度不同，将浮点型分为 3 种。

- ✧ 单精度浮点型：float。
- ✧ 双精度浮点型：double。



◇ 长双精度浮点型: long double。

### (3) 字符型

包括字符型常量和字符型变量。字符型常量中,一类重要的字符常量叫做转义字符,用来表示那些用一般字符不便于表示的控制代码。

字符型变量所占的内存空间为 1 字节。

### (4) 枚举类型

枚举类型是 ANSI C 新标准增加的数据类型。枚举类型是可以将变量所有可能取得的值一一列出的类型。应用枚举类型定义变量时要注意,变量的值只限于枚举元素的值的范围。

## 5. 运算符

C 语言中运算符丰富,共有 34 种。本章主要介绍 7 类运算符。

### (1) 算术运算符

基本的算术运算符有 5 种。

- ◇ +: 加法或正值运算符。它是二目运算符。
- ◇ -: 减法或负值运算符。它是二目运算符。
- ◇ \*: 乘法运算符。它是二目运算符。
- ◇ /: 除法运算符。它是二目运算符。
- ◇ %: 求余数运算符。它是二目运算符。

优先次序为按运算符的优先级别高低次序执行。例如:先乘除后加减,先括号里后括号外等。

自增自减运算符包括下列 4 种。

- ◇ ++i: i 自增 1 后再参与其他运算。
- ◇ --i: i 自减 1 后再参与其他运算。
- ◇ i++: i 参与运算后, i 的值再自增 1。
- ◇ i--: i 参与运算后, i 的值再自减 1。

强制转换运算符的形式如下:

(类型说明符)	(表达式)
(类型说明符)	x

### (2) 关系运算符

关系运算又叫做比较运算,是逻辑运算的一种简单形式。在 C 语言中提供了 6 种关系运算符。

- ◇ <: 小于。
- ◇ <=: 小于或等于。
- ◇ >: 大于。
- ◇ >=: 大于或等于。
- ◇ ==: 等于。



◇ !=: 不等于。

关系运算符的优先次序为<, <=, >, >=的优先级相同, 高于==和!=, ==和!=的优先级相同。

### (3) 逻辑运算符

在C语言中提供了3种逻辑运算符。

◇ &&: 与运算。

◇ ||: 或运算。

◇ !: 非运算。

逻辑运算符的优先次序为!(非)>&&(与)>||(或), 其中非运算(!)的优先次序最高。如果将赋值运算、逻辑运算、关系运算、算术运算放到一起时, 它们的优先次序是:

赋值运算符<&&和||<关系运算符<算术运算符<!(非)

赋值运算的优先级最小, 非运算的优先级最高。

### (4) 条件运算符

在一定条件下, 条件运算符可以代替条件语句进行条件判断和执行赋值。条件运算符组成条件表达式的一般形式为:

表达式1? 表达式2: 表达式3

### (5) 赋值运算符

赋值运算符为“=”, 赋值表达式的一般形式为:

变量=表达式  
变量=数值

### (6) 逗号运算符

逗号在C语言中的作用主要有两种: 一是作为分隔符使用; 二是作为语句中的运算符使用。

逗号作为运算符使用的情况一般有两种: 一是应用在for循环语句中; 二是将若干个表达式用逗号“,”连接, 组成一个逗号表达式。逗号表达式的一般形式为:

表达式1, 表达式2, ..., 表达式n

### (7) 求字节数运算符

求字节数运算符为sizeof, 它有两种用法:

sizeof 表达式

和

sizeof(类型)

在上一章中我们介绍了C语言的数据类型、运算符等相关知识。它们是程序设计的基本要素，是C程序设计的基础。本章主要介绍C语言的基本语句，通过对本章的学习，读者可以掌握编写C程序的基本方法，并可以编写出简单的C程序。

### 3.1 C 语句概述

计算机要实现人们预期的工作，就必须执行人们事先存入计算机内存中的指令，然后由计算机的中央处理器一条一条地执行每一条指令。这是典型的冯·诺依曼机的工作原理。而计算机能够唯一识别和执行的机器指令。也就是由0和1组成的机器编码。因此，应用任何语言（高级语言）编写的程序，都要编译或汇编成机器指令，计算机才能识别和执行。

作为高级语言的C程序，是由C语句构成的。一个C程序是由若干条C语句构成的，而每一条C语句最终又会编译成若干条机器指令。因此，无论我们的程序写成怎样，最终到计算机内部执行时就是一堆0、1机器代码。

一般来说，C语句都是能够完成一定操作任务的语句。因此，像前面提到的变量的定义、类型的声明以及库文件的包含都不属于C语句。而对于C程序中调用的函数，函数的声明部分不属于C语句，函数的执行部分由语句构成。

一个程序包括数据描述和数据操作两部分。数据描述部分由声明来实现，主要是定义数据类型和设置数据的初值，它不属于语句。数据操作部分由语句来实现，它的工作是按照程序员的要求对已提供的数据进行加工处理。

C语句共分为以下5类。

#### （1）控制语句

前面已经简要地介绍的if-else语句和for语句都属于控制语句。控制语句用于控制程序的流程，以实现程序的各种结构方式，由特定的语句定义符组成。C语言中只有9种控制语句。可分成以下三类。

- ◇ 分支语句：if语句、switch语句。
- ◇ 循环执行语句：do while语句、while语句、for语句。
- ◇ 转向语句：break语句、goto语句、continue语句、return语句。

每一种语句的具体使用方法在本章中会做详细介绍。



### (2) 函数调用语句

函数是程序中的功能模块，程序通过调用函数来实现一定功能。前面出现的 `printf()`、`scanf()` 等语句都属于函数调用语句。由一次函数调用加一个分号“;”构成的语句称为函数调用语句。函数调用语句的一般形式为：

函数名(实际参数表);

例如函数调用：

```
printf("Hello world!!");
```

### (3) 表达式语句

表达式语句是程序设计中应用最为广泛的语句。表达式语句由表达式加上分号“;”组成。它的一般形式为：

表达式;

例如表达式语句

```
x=y;      赋值语句;  
x=y+z;    加法运算语句, 计算赋值给 x;  
i++;      自增1语句, i 值增1。
```

### (4) 空语句

只有分号“;”组成的语句称为空语句。空语句表示什么也不做，在程序中空语句大多可用来作空循环体使用。

### (5) 复合语句

把多条语句用“{ }”括起来形成一条复合语句。在程序中，一条复合语句就是一条语句，而不要认为它是多条语句。例如：

```
{  
    x=y+z;  
    a=x*3;  
    printf("%d%d", x, a);  
}
```

就是一条复合语句。

另外，复合语句内的各条语句都必须以分号“;”结尾，在大括号“}”外不能加分号。而且复合语句的最后一个语句后面的分号不能省略。

## 3.2 C 程序的结构

从程序流程的角度来看，程序可以分为三种基本结构，即顺序结构、分支结构和循环结构。无论多么复杂的程序，都由这三种基本结构构成。一般地，顺序结构的程序主要由简单的表达式语句和函数调用语句组成，程序从上到下逐句执行。分支结构的程序主要由条件判断语句和转向语句组成，程序分叉执行。转移的条件包括条件转移和无条件转移。循环结构的程序主要由循环执行语句组成，程序按照规定的循环次数循环执行。

### 3.2.1 顺序结构

顺序结构是最简单的程序结构。计算机从程序的第一条语句开始执行，一直到程序的最后一条语句结束，中间没有分支跳转，也没有循环。顺序程序的结构如图3-1所示。

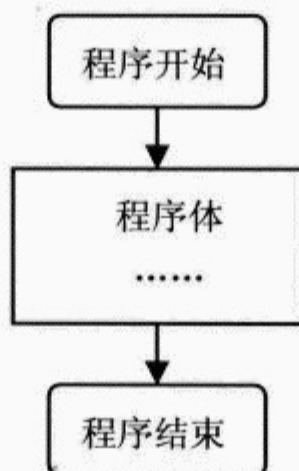


图 3-1 顺序程序的结构

下面通过例子来理解顺序程序。

#### 例程 3-1 顺序程序演示。

```
#include <stdio.h>
int main(void)
{
    int x,y,sum;
    printf("Please input the first integer\n");
    scanf("%d",&x);
    printf("Please input the second integer\n");
    scanf("%d",&y);
    sum=x+y;
    printf("The sum is:%d\n",sum);
    getchar();
    return 0;
}
```

本例程是一个典型的顺序结构程序，计算机按照程序的顺序逐条执行每一条语句。直到主程序返回 `return 0` 结束。其实，在计算机内部，每条 C 语句都会被编译成为若干条机器代码，这些机器代码按照编译系统分配的内存地址存放在内存中，然后由中央处理器（CPU）逐条取出，顺序执行。本例程的运行结果是：

```
Please input the first integer
5
Please input the second integer
10
The sum is:15
```

### 3.2.2 分支结构

分支结构的程序在程序执行时并不是一成不变地从头执行到尾，而是根据不同的条件执行不同路径的程序段，在程序执行时发生了“分叉”、“跳转”。常常是应用条件判断语句来决定程序的走向，也有无条件转移的情况。分支程序的结构如图3-2所示。



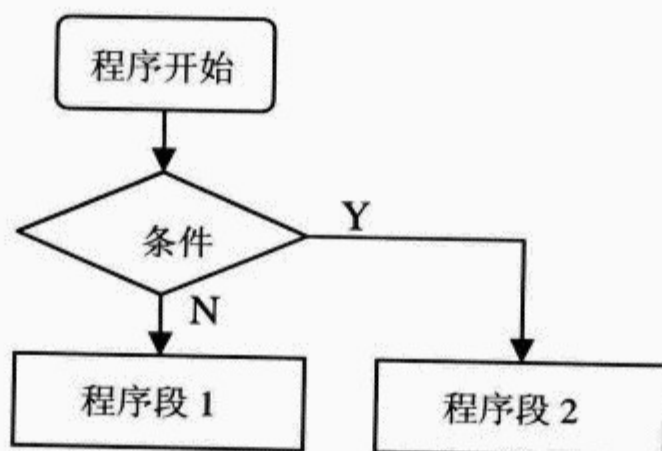


图 3-2 分支程序的结构

图 3-2 中的条件一般是条件判断语句 (if 语句, switch 语句)。如果符合条件的转移要求, 就执行程序段 2, 如果不符合条件的转移要求, 就顺序执行程序段 1。

下面通过例子来理解分支程序。

### 例程 3-2 分支程序演示。

```

#include <stdio.h>
int main(void)
{
    char c;
    c=getchar();
    if(c>='a'&&c<='z' || c>='A'&&c<='Z')
        printf("Is alpha\n");
    else
        printf("Other\n");
    getchar();
    return 0;
}
  
```

本例程是一个典型的分支结构程序, 作用是判断输入的字符是否为字母, 如果是字母, 则在屏幕上显示 "Is alpha" 字符串提示; 如果不是字母, 则显示 "Other" 提示。判断是不是字母的条件是逻辑表达式 `c>='a'&&c<='z' || c>='A'&&c<='Z'`, 通过条件判断语句 (if 语句) 对输入的字符进行判断, 并根据逻辑表达式 `c>='a'&&c<='z' || c>='A'&&c<='Z'` 的真值决定程序下一步执行哪条语句。本例程的运行结果为:

```

2
Other
d
Is alpha
T
Is alpha
  
```

### 3.2.3 循环结构

许多问题的解决不是单靠执行一遍程序就可以完成的, 而是需要重复执行相同类型的操作数次, 数十次, 甚至成百上千次才能完成, 这就需要用到循环控制。比如计算式子  $1+2+\dots+10000$ , 就需要用到循环累加。因此, 进行 C 程序设计, 循环控制是必不可少的。循

环程序的基本结构如图3-3所示。

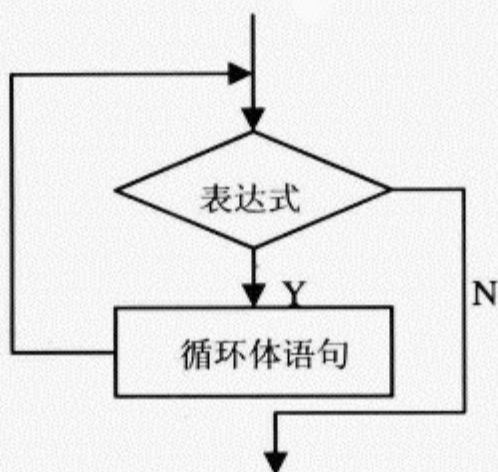


图 3-3 循环程序的结构

图 3-3 中的表达式一般是关系表达式或逻辑表达式。当表达式的值为 1 时，程序保持循环状态，即执行循环体语句后返回到表达式上面的语句，然后再通过表达式判断；如果表达式的值为 0，则跳出本次循环。

下面通过例子来理解循环程序。

### 例程 3-3 循环程序演示。

```
#include <stdio.h>
int main(void)
{
    int i,sum=0;
    for(i=1;i<=100;i++)
        sum=sum+i;
    printf("The result of 1+2+...+100 is\n%d",sum);
    getchar();
    return 0;
}
```

本例程通过 for 循环语句计算式子  $1+2+\dots+100$  的值。这里用关系表达式  $i \leq 100$  作为判断程序是否进行循环操作的条件。变量 sum 是累加变量，用来存放计算结果。本例程的运行结果为：

```
The result of 1+2+...+100 is
5050
```

以上是 C 程序的三种基本结构。当然，在一个程序中不可能只有单一的一种程序结构，它可能是几种结构的综合。但是，无论编写的程序多么复杂，都要以这三种基本结构为基础。因此，学好这三种基本结构十分重要。下面将详细地介绍 C 语言的赋值语句、分支语句和循环语句，用这三种语句就可以实现 C 程序的三种基本结构。

## 3.3 基本的赋值语句

### 1. 赋值运算符

赋值运算符用 “=” 表示，它的作用是将一个数值赋值给一个变量。用 “=” 连接的式



子称为赋值表达式。其一般形式为：

变量=表达式

例如：

```
x=5;  
a=b;  
x=y+z
```

都是合法的赋值表达式。上面第一个式子的意思是将 5 赋值给变量 x；第二个式子的意思是将变量 b 中的值赋值给变量 a；第三个式子的意思是将变量 y 和 z 中的值求和，再将结果赋值给变量 x。

赋值运算符具有右结合性。即运算的次序是从右向左的。因此赋值表达式

a=b=c=d

就等价于

a=(b=(c=d))

赋值表达式作为表达式的一种，具有表达式的一般特性。因而凡是表达式可以出现的地方均可出现赋值表达式。第 2 章中曾介绍过，if 语句中 if 关键字的后面要有表达式，一般是关系表达式或逻辑表达式：

```
if(关系表达式)  
if(逻辑表达式)
```

但是，出现赋值表达式也是合法的。例如：

```
if(a=1)b++;
```

有关 if 语句的使用下面还会详细介绍。

如果赋值表达式再加上分号“;”就构成了赋值语句，赋值语句即可以独立执行。

## 2. 类型转换

如果赋值运算符两边的数据类型不相同，系统将自动进行类型转换，即把赋值号右边的类型换成左边的类型。具体规定如下：

(1) 浮点型赋值给整型，舍去小数部分。

(2) 整型赋值给浮点型，数值不变，但将整型数以浮点形式存放，即增加小数部分（小数部分的值为0）。

(3) 字符型赋值给整型，由于字符型为1字节，而整型为2字节，故将字符的ASCII码值放到整型量的低八位中，高八位为0。

(4) 整型赋值给字符型，只把低八位赋值给字符型变量，高八位舍去。

## 3. 复合的赋值运算符

在赋值符“=”之前加上其他二目运算符可构成复合赋值符。如+=，-=，\*=，/=，%=，<<=，>>=，&=，^=，|=。

构成复合赋值表达式的一般形式为：

变量 双目运算符=表达式

它等效于

变量=变量 运算符 表达式

例如：

$a+=1$	等价于 $a=a+1$
$x*=y+2$	等价于 $x=x*(y+2)$
$a\%=b$	等价于 $a=a\%b$

在编程时提倡广泛应用复合赋值表达式。因为它不但使得代码简洁、清晰，而且有利于源程序的编译处理，从而能提高编译效率并产生质量较高的目标代码。

## 3.4 分支语句和循环语句

利用分支语句可以构成分支结构的程序。分支语句的作用是根据指定的条件，从给定的两组（或多组）分支操作中选择其一。分支语句主要包括以下两种：

- ✧ if 语句（二选一）。
- ✧ switch语句（多选一，开关语句）。

利用循环语句可以构成循环结构的程序。许多问题的解决需要重复执行相同类型的操作。人类之所以发明计算机，就是要应用计算机快速的计算能力。因此，循环语句在各种编程语言中都不可缺少。C语言中提供了4种循环语句，分别为：

- ✧ for语句。
- ✧ while语句。
- ✧ do-while语句。
- ✧ goto语句。

在下面的小节中将分别介绍这两种分支语句和这4种循环语句。

## 3.5 if 语句

if语句通过判断给定的条件（一般是关系表达式或逻辑表达式）是否为真，从两组分支操作中选择其中一组分支操作。C语言提供了3种形式的if语句。

### 3.5.1 第一种形式的if语句

第一种if语句的基本形式为：

if(表达式) 语句

这里的语句既可以是简单语句，也可以是复合语句。例如：



```
if(a>b)printf("%d",a);
```

或

```
if(a>b){  
    a++;  
    b--;  
}
```

if 语句的执行过程是：如果表达式为真（即真值为 1），则执行语句，否则继续执行后续程序。

其过程可表示为图3-4。

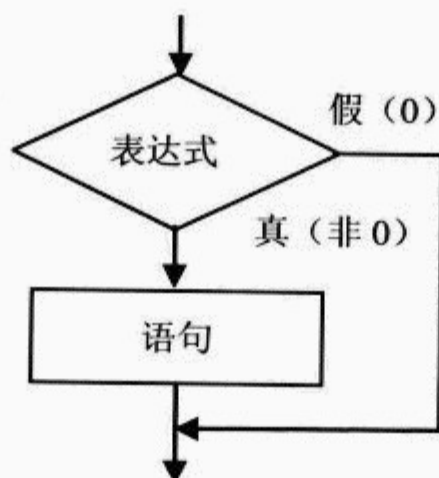


图 3-4 第一种 if 语句执行过程

### 3.5.2 第二种形式的 if 语句

第二种 if 语句的基本形式为：

```
if(表达式)  
    语句 1;  
else  
    语句 2;
```

该语句的执行过程是：如果表达式为真（即真值为 1），则执行语句 1，否则执行语句 2。其过程可表示为图 3-5。

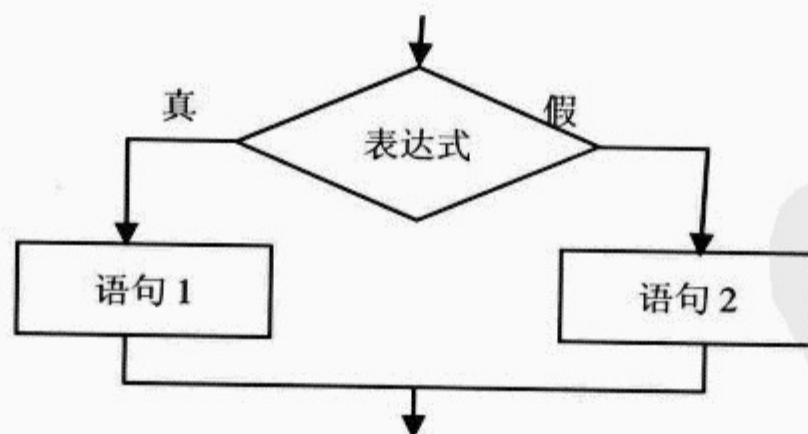


图 3-5 第二种 if 语句执行过程

### 3.5.3 第三种形式的 if 语句

第三种 if 语句的形式可以实现多分支选一，它是第二种 if 语句的形式的扩展。其一般

形式为：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
else if(表达式 3)
    语句 3;
...
else if(表达式 m)
    语句 m;
else
    语句 n;
```

该语句的执行过程是：依次判断表达式的值，当出现某个值为真时，则执行其对应的语句，然后跳到整个 if 语句之外继续执行程序。如果所有的表达式均为假，则执行语句 n。然后继续执行后续程序。该语句的执行过程如图 3-6 所示。

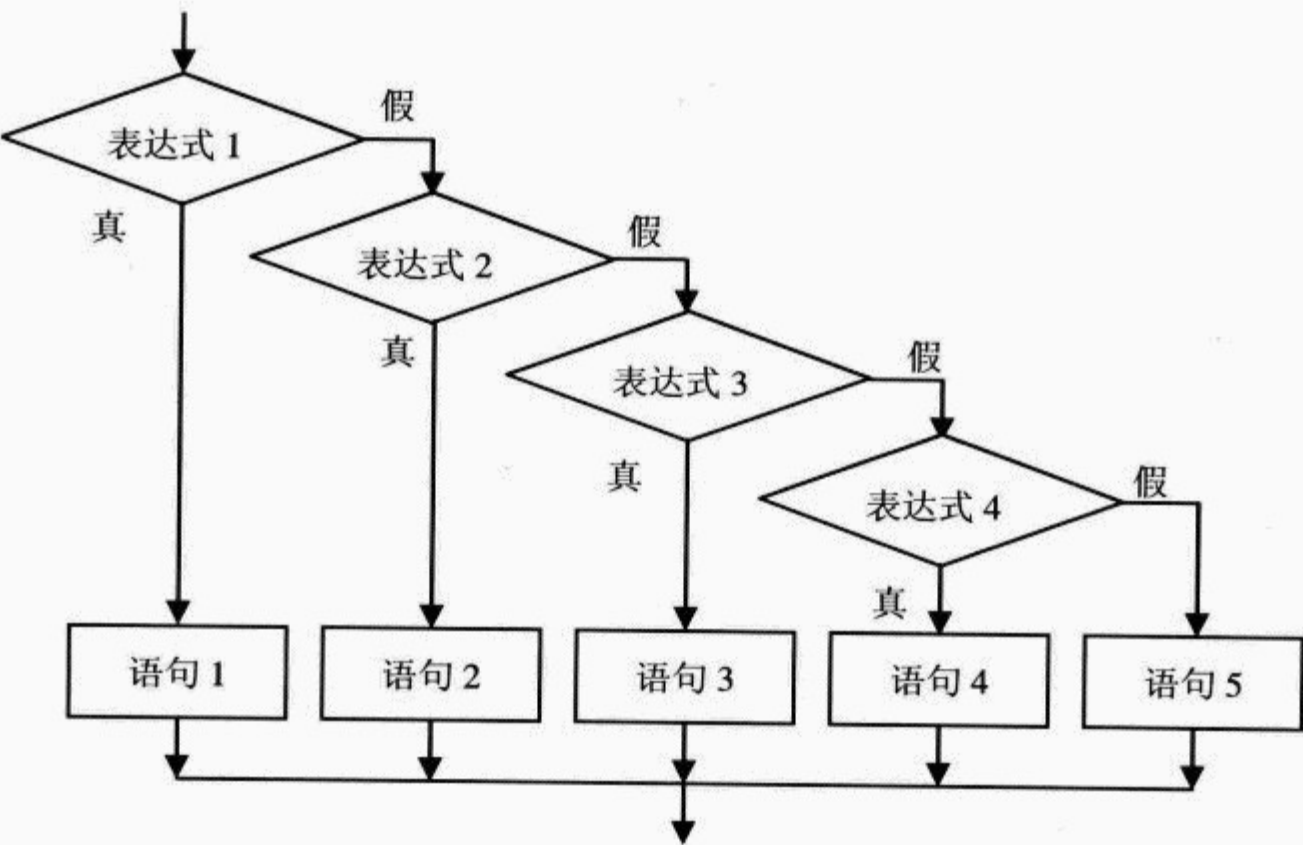


图 3-6 第三种 if 语句执行过程

3.5.4 三种 if 语句的程序举例

下面通过具体的例子来理解上述三种if语句。

例程 3-4 第一种 if 语句。

```
#include <stdio.h>
int main(void)
{
    int x,y,max;
    printf("Input two numbers:\n ");
    scanf("%d%d",&x,&y);
    max=x;
    if (max<y) max=y;
    printf("max=%d",max);
    getchar();
}
```



```
    return 0;
}
```

本例程首先提示用户输入两个数字，并将它们分别赋值给变量  $x$ ,  $y$ 。然后将变量  $x$  赋值给  $max$ ，也就是假设  $x$  最大。再将变量  $max$  同  $y$  比较，如果表达式  $max < y$  成立，即真值为 1，则说明  $max$  小于  $y$ ，即  $y$  值最大，于是执行语句  $max=y$ ；否则，顺序执行后续程序。最后输出比较结果。输入数字 2, 3 后本例程的执行结果为：

```
Input two numbers:
2
3
max=3
```

### 例程 3-5 第二种 if 语句。

```
#include <stdio.h>
int main(void)
{
    int x,y,max;
    printf("Input two numbers:\n ");
    scanf("%d%d",&x,&y);
    if (x<y)
        max=y;
    else
        max=x;
    printf("max=%d",max);
    getchar();
    return 0;
}
```

本例程与上例的功能完全相同，只是使用了第二种 if 语句。如果  $x$  小于  $y$ ，则执行语句  $max=y$ ；否则执行语句  $max=x$ ；最终输出比较结果。

### 例程 3-6 第三种 if 语句。

```
#include <stdio.h>
int main(void)
{
    char c;
    printf("input a character: ");
    c=getchar();
    if(c<32)
        printf("This is a control character\n");
    else if(c>='0'&&c<='9')
        printf("This is a digit\n");
    else if(c>='A'&&c<='Z')
        printf("This is a capital letter\n");
    else if(c>='a'&&c<='z')
        printf("This is a small letter\n");
    else
        printf("This is an other character\n");
    getchar();
    return 0;
}
```

本例程首先提示用户输入一个字符，然后应用第三种 if 语句对输入的字符进行判断。根据输入字符的 ASCII 码范围选择执行语句的分支。

如果输入字符的 ASCII 码小于 32，则表明该字符是控制字符；如果输入字符的 ASCII 码

在字符'0'~'9'之间,则表明该字符是数字字符;如果输入字符的ASCII码在字符'A'~'Z'之间,则表明该字符是大写字母字符;如果输入字符的ASCII码在字符'a'~'z'之间,则表明该字符是小写字母字符;否则,表明该字符是其他字符。本例程的运行结果为:

输入数字 3

```
input a character: 3
This is a digit
```

输入字符 a

```
input a character: a
This is a small letter
```

输入字符 A

```
input a character: A
This is a capital letter
```

输入字符 &

```
input a character: &
This is an other character
```

### 3.5.5 有关 if 的一些说明

在上述三种形式的 if 语句中,if 关键字的后面均为表达式,并用括号括起来。该表达式一般情况下为逻辑表达式或关系表达式,但也可以是其他表达式,像赋值表达式等,甚至还可以是一个变量。只要表达式的值非 0 即为“真”,表达式的值为 0 即为“假”。例如:

```
if(a=1) 语句;
if(1) 语句;
if(a) 语句;
```

括号中的表达式分别为赋值表达式、常量、变量,它们都有值。像 `a=1`,如果赋值成功,则该赋值表达式的值为 1;在 `if(1)` 中,括号里是常量 1,因此为真;在 `if(a)` 中,如果 `a` 的值非 0,则为真,否则为假。

另外,在上述三种形式的 if 语句中,所有的语句应为单个语句,如果要想在满足条件时执行多个语句,则必须把这一组语句用“{}”括起来组成一个复合语句。但要注意在“}”之后不能再加分号。例如:

```
if(a>b)
{
    max=a;
    min=b;}
else
{
    max=b;
    min=a;
}
```

### 3.5.6 if 语句的嵌套

上面讲到了三种 if 语句的形式,如果将上述三种 if 语句形式中的“语句”部分换为 if 语句,也就是当 if 语句中的执行语句为 if 语句时,就构成了所谓 if 语句的嵌套。其一般形



式可表示为:

```
if(表达式)
    if ( ) 语句1
    else 语句2
```

或者为:

```
if(表达式)
    if ( ) 语句1
    else 语句2
else
    if ( ) 语句1
    else 语句2
```

这里需要讨论一下 if 和 else 的配对问题。在 C 语言中, 为了避免程序的二义性, C 语言规定 else 总是与它前面最近的 if 配对。例如:

```
if(表达式1)
    if(表达式2)
        语句1;
    else
        语句2;
```

程序中, else 是与第一个 if 配对还是与第二个 if 配对呢? 根据 C 语言的规定, else 是与第二个 if 配对的。也就是说, 第二个 if 与 else 构成的语句作为第一个 if 的语句。它完全等价于:

```
if(表达式1)
{
    if(表达式2)
        语句1;
    else
        语句2;
}
```

因此, 在分析复杂的 if 语句时, 先要弄清哪个 if 与哪个 else 配对, 这样理解程序时才不会出错。

下面通过例子来理解 if 语句的嵌套使用。

### 例程 3-7 学校进行成绩分级管理, 取消分数制, 改为成绩分级评定。

具体办法是: 小于60分为“E”类; 60分至70分(不含70分)为“D”类; 70分至80分(不含)为“C”类; 80分至90分(不含)为“B”类; 90分以上为“A”类。设计一个程序, 对输入的成绩进行等级划分。

分析: 如果用第三种 if 语句进行判断是可以的。即:

- (1) 先看输入的成绩是否小于60, 是则为“E”类, 否则(2)。
- (2) 再看输入的成绩是否小于70, 是则为“D”类, 否则(3)。
- (3) 再看输入的成绩是否小于80, 是则为“C”类, 否则(4)。
- (4) 再看输入的成绩是否小于90, 是则为“B”类, 否则(5)。
- (5) 该成绩为“A”类。

用程序流程图表示为图 3-7。

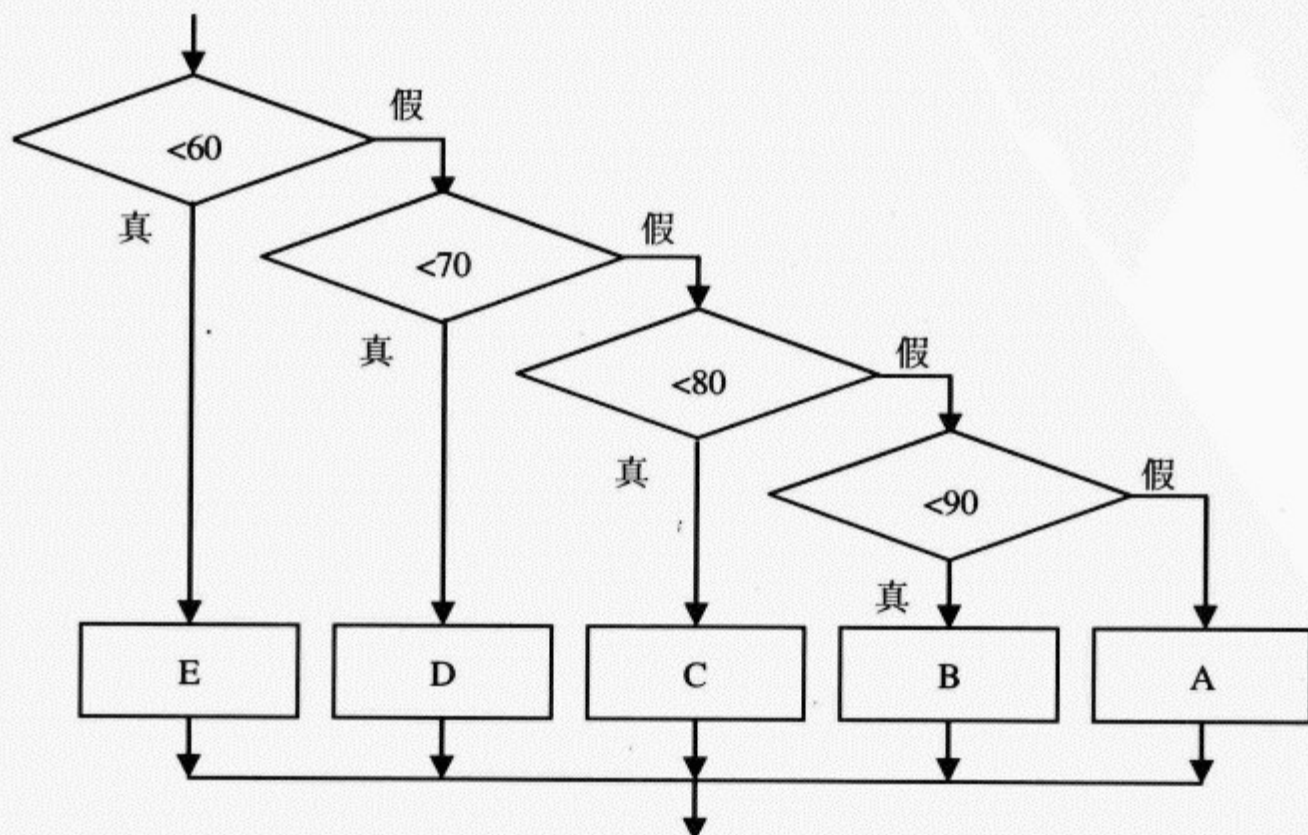


图 3-7 应用第三种 if 语句的执行过程

通过该程序执行的流程图可以看出，如果一个学生的成绩为95分，那他至少要在该程序中进行4次比较才能最终得到结果。倘若同学们的成绩普遍很好，那么用这个程序进行成绩等级划分的效率是不高的，因为它最多要比较4次才能得到最终结果。如果应用if语句的嵌套，就可能减少比较次数，从而提高程序的效率。可以设计这样一个程序执行的流程，如图3-8所示。

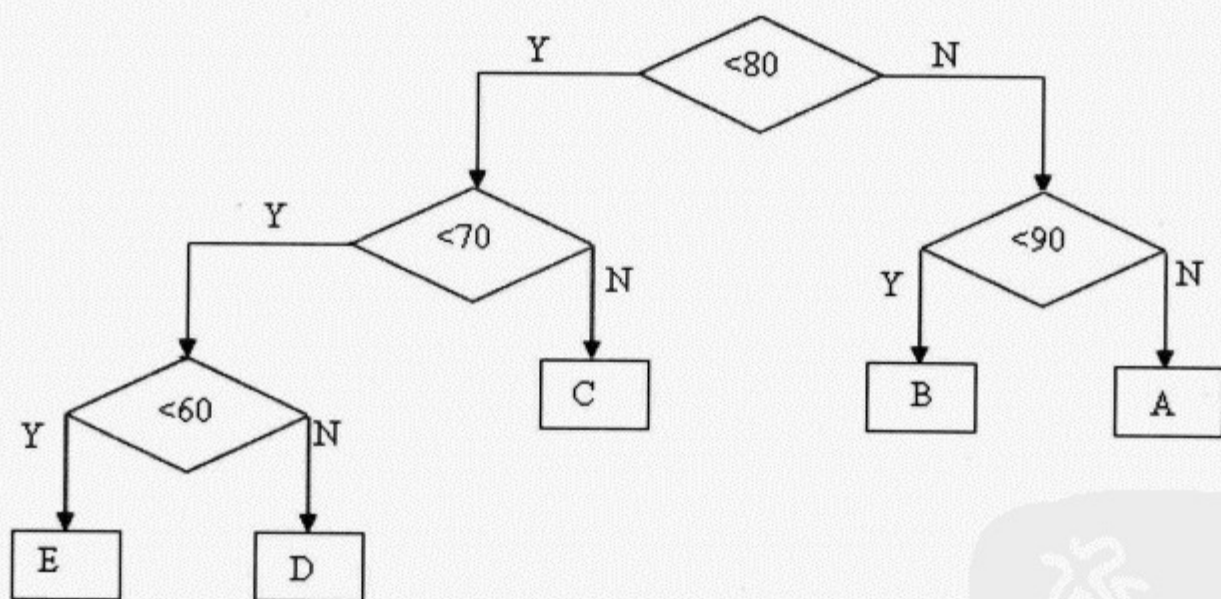


图 3-8 应用 if 嵌套语句的一种执行流程

从流程图中不难看出，应用 if 嵌套语句可以减少程序的平均比较次数。因为它最多只要比较 3 次就能得到最终结果。同样是成绩普遍很好的同学，应用这个程序来进行成绩等级划分的效率就要比图 3-7 所示高。如果是在大型系统中，需要比较的次数更多，那么这个程序的优势将更加明显。落实到代码上如下：

```
#include <stdio.h>
int main(void)
```



```
{
    int score;
    printf("Please input the score\n");
    scanf("%d",&score);
    if(score<80)
        if(score<70)
            if(score<60)
                printf("E\n");
            else
                printf("D\n");
        else
            printf("C\n");
    else
        if(score<90)
            printf("B\n");
        else
            printf("A\n");
    getchar();
    return 0;
}
```

假设几个同学的成绩分别为 59、68、72、80、91，通过本程序得到的等级标准分别是：

```
Please input the score
59
E
```

```
Please input the score
68
D
```

```
Please input the score
72
C
```

```
Please input the score
80
B
```

```
Please input the score
91
A
```

## 3.6 switch 语句

除了上面介绍的if语句，C语言还提供了另一种用于多分支选择的switch语句。if语句也可以用于多分支，但是容易混乱，特别是if的嵌套，如果使用得不当就会产生表达上的错误，而且使得程序的可读性大大降低。因此建议在仅有两个分支或少数分支时使用if语句，而在有多个分支时使用switch语句。



### 3.6.1 switch 语句的一般形式

switch 语句的一般形式为:

```
switch(表达式){
    case 常量表达式 1: 语句 1;
    case 常量表达式 2: 语句 2;
    ...
    case 常量表达式 n: 语句 n;
    default           : 语句 n+1;
}
```

程序表达的是计算表达式的值,并逐个与其后的常量表达式的值比较,当表达式的值与某个常量表达式的值相等时,就执行它后面的语句,然后就不再进行判断,继续执行后面所有 case 后的语句了。如果表达式的值与所有 case 后的常量表达式均不相同,则执行 default 后的语句。

必须注意的是,这里的“case常量表达式”在switch语句中只是分支的入口,并不是在该处进行条件判断。因此,一旦表达式与case后面的常量表达式匹配成功,就从这个入口处执行下去,然后就不再进行判断,继续执行后面所有case后的语句了。这样switch语句不能真正起到多分支选择的作用。我们可以通过例子理解这一点。

#### 例程 3-8 未经改造的 switch 语句。

```
#include <stdio.h>
int main(void)
{
    int x;
    printf("Input integer number: ");
    scanf("%d",&x);
    switch (x){
        case 1:printf("Monday\n");
        case 2:printf("Tuesday\n");
        case 3:printf("Wednesday\n");
        case 4:printf("Thursday\n");
        case 5:printf("Friday\n");
        case 6:printf("Saturday\n");
        case 7:printf("Sunday\n");
        default:printf("error\n");
    }
    getchar();
    return 0;
}
```

本例程中提示输入一个整型数,根据数值对应其一周的星期数。如前面所讲,当输入一个整数并与 case 后面的常量表达式匹配成功时,就执行它后面的语句,然后就不再进行判断,继续执行后面所有 case 后的语句。因此,如果输入 3,该程序的输出结果不是简单的 Wednesday,而是:

```
Input integer number: 3
Wednesday
Thursday
Friday
Saturday
Sunday
error
```



### 3.6.2 带有 break 语句的 switch 语句

上一节中switch语句的结果不是我们希望得到的，因为它不能真正起到分支语句的作用。因此要对上述的switch语句进行改造，使它能够输出正确的结果，具备分支语句的功能。

现在希望的是执行完一个case分支后，使流程马上跳出switch语句，这样就不会再执行后面所有case后的语句了。当然，如果表达式的值与所有case后的常量表达式均不相同，还要执行default后的语句。可以结合使用break语句来实现这个程序跳出的功能。参看例程3-9。

**例程 3-9** 带有 break 语句的 switch 语句。

```
#include <stdio.h>
int main(void)
{
    int x;
    printf("input integer number:");
    scanf("%d",&x);
    switch (x){
        case 1:printf("Monday\n");break;
        case 2:printf("Tuesday\n"); break;
        case 3:printf("Wednesday\n");break;
        case 4:printf("Thursday\n");break;
        case 5:printf("Friday\n");break;
        case 6:printf("Saturday\n");break;
        case 7:printf("Sunday\n");break;
        default:printf("error\n");
    }
    getchar();
    return 0;
}
```

在本例程中，当输入的整数与 case 后面的常量表达式匹配成功，就执行 case 后面的语句，然后执行 break 语句，强制跳出 switch 语句。这样就不会像上面的例子那样执行所有的 case 后面的语句了。注意，default 后面可以不加 break，因为这是到了 switch 语句最后，加与不加效果是一样的。

本例程的运行结果为：

当输入 5 时

```
input integer number:5
Friday
```

当输入 8 时

```
input integer number:8
error
```

### 3.6.3 有关 switch 语句的一些说明

使用 switch 语句时要注意的几点：

- (1) case 后各常量表达式的值不能相同，否则会出现错误。
- (2) case 后允许有多个语句，可以不用“{ }”括起来，程序会顺序执行。例如：

```
Case 'A':printf("ASCII is 65");count++;break;
```



(3) 各 case 和 default 子句的先后顺序可以变动, 而不会影响程序执行结果。但要注意, 如果 default 子句前置, 后边要加 break 语句结果才正确, 只有最后的分支语句可以不加 break 而不影响结果。

(4) default 子句也可以省略不用。

(5) switch 的参数不能是浮点型, case 后面必须是整型数或者整型表达式。例如:

```
switch(x){case 1.0:i++;case 2.0:i--;}
```

这样的用法是错误的。

## 3.7 for 语句

for 语句是十分常用的一种循环语句, 也是 C 语言中最为灵活的一种循环语句。在前面的章节中已经简要地介绍了 for 语句的用法, 这里将进一步说明。

### 3.7.1 for 语句的一般形式

for 语句一般形式为:

```
for(表达式1; 表达式2; 表达式3) 语句
```

其中, 语句既可以是简单的语句, 又可以是用 “{}” 括起来的复合语句。for 语句的执行过程为:

(1) 先求解表达式1。

(2) 求解表达式2, 若其值为真 (非0), 则执行 for 语句中指定的内嵌语句, 然后执行下面第 (3) 步; 若其值为假 (0), 则结束循环, 转到第 (5) 步。

(3) 求解表达式3。

(4) 转回上面第 (2) 步继续执行。

(5) 循环结束, 执行 for 语句下面的一个语句。

我们可以先用一个简单的例子来熟悉一下 for 语句的执行过程。

#### 例程 3-10 简单的 for 循环语句。

```
#include <stdio.h>
int main(void)
{
    int i, sum=0;
    for(i=1; i<=100; i++)
        sum=sum+i;
    printf("%d\n", sum);
    getchar();
    return 0;
}
```

这是一个求解  $1+2+\dots+100$  之和的程序, 应用 for 语句实现求解过程。具体的步骤如下:



(1) 先求解表达式1, 即将1赋值给i。

(2) 判断 $i \leq 100$ 的真值, 即i是否不大于100, 若是, 则计算表达式 $\text{sum} = \text{sum} + 1$ ; sum是个累加变量, 最终将 $1+2+\dots+100$ 的和存放到sum中。然后再执行第(3)步。若i大于100, 转到执行第(5)步。

(3) 求解表达式3, 即 $i++$ 。

(4) 回到第(2)步继续执行: 判断 $i \leq 100$ 的真值。

(5) 循环结束, 执行for语句下面的一个语句, 即显示计算结果。

以上就是 for 语句执行的整个流程。可以用流程图 3-9 形象地表示这一过程。

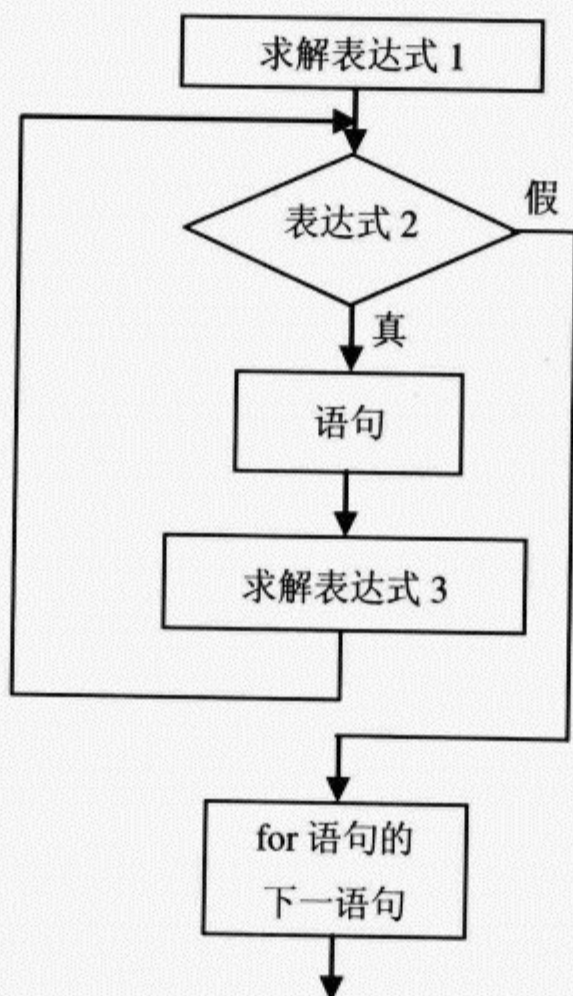


图 3-9 for 语句执行过程

### 3.7.2 有关 for 语句的一些说明

在应用 for 语句编写循环结构的程序时, 应当注意以下几点:

(1) for语句中的“表达式1”、“表达式2”和“表达式3”都是选择项。也就是说可以默认, 但分号“;”不能默认。

(2) 如果省略了“表达式1”, 表示不对循环控制变量赋初值, 应当在for语句之前给循环变量赋初值。因此“表达式1”又叫做循环变量赋初值(上例中的i称为循环变量)。

(3) 如果省略了“表达式2”, 便产生死循环。死循环就是在不做其他处理的情况下永不停地循环下去。因此“表达式2”又叫做循环条件。

例如:

```
for(i=1;;i++) sum=sum+i;
```



就是一个死循环，不做其他处理的情况下，它将不会停止运算。

(4) 如果省略了“表达式3”，就不会对循环控制变量进行操作，这时通常可在语句体中加入修改循环控制变量的语句。因此“表达式3”又叫做循环变量增量或步长值。

例如可将上例改为：

```
for(i=1;i<=100;)
{sum=sum+i;
i++;}
```

效果是一样的。

(5) 表达式1可以是设置循环变量的初值的赋值表达式，也可以是其他表达式。

例如：

```
for(sum=0;i<=100;i++)sum=sum+i;
```

其中，i是循环变量，sum不是循环变量。

(6) 表达式1和表达式3可以是一个简单表达式，也可以是逗号表达式。

例如：

```
for(sum=0,i=1;i<=100;i++)sum=sum+i;
```

(7) 表达式2一般是关系表达式或逻辑表达式，但也可以是其他表达式，只要其值非零，就执行循环体。

例如：

```
for(i=0;(c=getchar())!='\n';i+=c);
```

上述例子的循环条件是输入一个不是换行符的字符，即如果输入的不是换行符，就一直循环下去，直到输入换行符为止。

### 3.7.3 for 语句程序举例

下面通过例子来深入理解for循环语句。

#### 例程 3-11 判断一个正整数是否是质数。

分析：质数就是指除了1和它本身之外，不能被任何数整除的数。像1、2、3、5、7等都是质数。如果要判断一个数(a)是否是质数，最简单的方法就是应用循环判断，即将该数(a)分别被2、3……a-1求模。在这个过程中，如果求模的结果全为非0，则表明该数都不能被除了1和它本身之外的数整除，即该数一定是质数；否则，一旦出现求模的结果为0时，就可以结束循环，表明该数可以被除了1和它本身之外的数整除，也就是说该数不是质数。比如整数6， $6\%2=0$ ，因此6不是质数。本例程代码如下：

```
#include <stdio.h>
int main(void)
{
    int a,i ;
    printf("Please input a number:\n");
    scanf("%d",&a);
    for(i=2;i<a;i++){
        if(a%i==0)
            break;
    }
```



```

}
if(i==a||1==a)
printf("%d is a prime number\n",a);
else
printf("%d is not a prime number\n",a);
getchar();
return 0;
}

```

该程序中， $i$  是循环变量， $i$  从 2 开始被  $a$  求模，直到  $i$  的值累加到  $a-1$  为止。在此期间，如果  $a\%i==0$ ，就表明  $a$  不是质数，于是用 `break` 语句跳出本次循环，否则继续循环下去，直到  $i$  的值为  $a-1$  为止。然后再判断  $i$  是否与  $a$  相等，如果相等则说明 `for` 循环是自然结束而非强制跳出，因此  $a$  是质数；否则说明循环是通过 `break` 语句强制跳出的，因此  $a$  不是质数。

这里在判断  $a$  是否是质数的条件上又加了一个  $1==a$ ，这是因为 1 虽然是质数，但在该程序中无法判断出来。原因是循环变量的初值为 2，1 不可能等于  $i$ 。因此要加上  $1==a$  判断条件予以修正。本例程的运行结果为：

```

Please input a number:
9
9 is not a prime number
Please input a number:
5
5 is a prime number

```

### 3.8 while 语句

前面介绍了 `for` 循环语句，在 C 语言中 `while` 循环语句也是一种十分常用的循环语句。它可以与一些语句配合实现 `for` 语句的功能。同时，`while` 语句容易理解，写成代码可读性强，因此使用十分广泛。

`while` 语句的一般形式为：

```
while(表达式) 语句
```

执行过程是：计算表达式的值，当值为真（非 0）时，执行循环体语句。可以用流程图图 3-10 形象地表示这一过程。

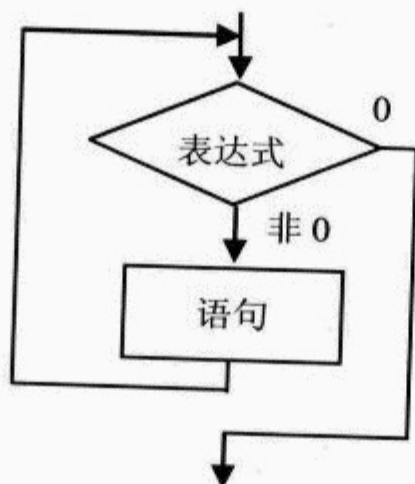


图 3-10 while 语句的执行过程

在使用 `while` 语句时应当注意以下几点：

(1) 上面给出的一般形式中, 语句既可以是简单的语句, 也可以是用“{}”括起来的复合语句。

(2) 上面给出的一般形式中的表达式一般是关系表达式或者逻辑表达式, 用以限定循环的次数。但也可以是其他表达式, 例如赋值表达式等, 甚至还可以是一个变量。只要表达式的值非0即为“真”, 循环继续; 表达式的值为0即为“假”, 循环停止。

下面通过例子来理解 while 语句的用法。

### 例程 3-12 while 语句的用法。

```
#include <stdio.h>
int main(void)
{
    int i, sum=0;
    i=1;
    while(i<=100)
    {
        sum=sum+i;
        i++;
    }
    printf("%d\n", sum);
    getchar();
    return 0;
}
```

本例程仍然是求  $1+2+\dots+100$  的值, 只是在这里应用了 while 语句。首先对变量  $i$  赋值 1, 然后进入 while 循环, 每次循环都执行循环体语句  $sum=sum+i$  和  $i++$ 。sum 是累加变量, 用来存放计算结果,  $i$  是循环变量, 用来控制循环次数。直到  $i$  大于 100 时, 即  $i$  等于 101 时, 该循环结束。 $i$  作为循环变量, 每次将其值累加到 sum 上, 最后 sum 的值就是  $1+2+\dots+100$  的结果。本程序的运行结果为:

5050

## 3.9 do-while 语句

do-while 语句与 while 语句类似, 只是执行循环体语句的顺序不同。while 语句是先通过表达式判断, 看是否满足循环条件, 如果满足条件才能进入循环, 即执行循环体语句。而 do-while 语句则是先执行循环体语句, 再判断循环条件是否成立。它的一般形式为:

```
do
    语句
while(表达式);
```

这样, 先执行循环体语句, 再进行循环判断。如果表达式值为真, 则循环继续; 如果表达式值为假, 则循环结束。因此, do-while 循环至少要执行一次循环语句。有时, 在进入循环之前必须要有一个终端输入, 然后根据这个输入来判断是否进行下一次循环, 这种情况下就可以使用 do-while 语句。



可以用流程图3-11形象地表示这一过程。

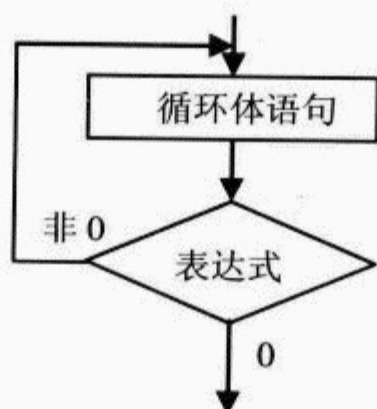


图 3-11 do-while 语句的执行过程

do-while语句可以代替while语句处理一些问题。但由于do-while语句的特点，有时应用do-while语句解决问题要比while语句更加方便、容易且程序可读性强。下面对while语句和do-while语句做一些比较。

### 例程 3-13 while 语句和 do-while 语句的比较。

#### (1) 用 while 语句：

```
#include <stdio.h>
int main(void)
{
    char c;
    c=getchar();
    while(c!=EOF){
        c=getchar();
    }
    getchar();
    return 0;
}
```

#### (2) 用 do-while 语句：

```
#include <stdio.h>
int main(void)
{
    char c;
    do
        c=getchar();
    while(c!=EOF);
    getchar();
    return 0;
}
```

以上两段程序分别用 while 语句和 do-while 语句实现同一个功能，即：用键盘向终端输入字符，直到输入 Ctrl+Z 结束符为止。但是应用 do-while 语句就显得比用 while 语句更加简练，程序的可读性更强。具体分析用两语句时程序的执行情况：

用while语句时，先要向变量c输入一个字符，然后进入while循环。循环条件是c!=EOF，EOF是结束标志，用组合键Ctrl+Z表示。也就是说直到输入Ctrl+Z结束符循环才结束。这里在循环语句外先要有一条c=getchar()语句，而循环体语句依然是c=getchar()，因此显得麻烦。如果代码复杂，程序的可读性就不强了。

用do-while语句时，先执行循环体语句c=getchar()，然后再进行循环判断。这样就将while语句时的两条c=getchar()语句合并为一条，使得代码更加简洁，程序的可读性增强。

## 3.10 goto 语句

goto语句又叫做无条件转移语句。与前面所讲的循环语句不同，goto语句本质上属于转向语句，但它在程序设计中往往与“标号”构成循环语句，能起到无条件循环的作用。使用goto语句进行循环操作时不需要任何条件，也就是不需要循环判断，因此将goto语句放到循环语句中来讨论。

goto 语句的一般形式是：

```
goto 语句标号;
```

它的意思是直接将程序流程跳转到“语句标号”标记的位置。例如：

```
loop:
...
goto loop;
...
```

当程序执行到 goto loop 时，会自动将程序流程转移到 loop 标记的位置，这样就无条件地产生了一个循环。

这里必须注意，语句标号是一个有效的标识符，因此它的命名规则和变量的命名规则一样，都要满足标识符格式要求。例如：

```
goto loop;
goto label;
```

等都是合法的。而

```
goto 123;
goto _loop;
```

等都不合法。

另外，语句标号在程序中出现时后面要加“;”，例如，loop: 当执行goto语句后，程序将跳转到该标号处并执行其后的语句。同时，语句标号必须与goto语句同处于一个函数中（例如主函数main），但可以不在一个循环层中。

在结构化的程序设计中不建议过多使用 goto 语句，滥用 goto 语句会使程序层次不清，且不易读。一般在两种情况下可以使用 goto 语句：

- (1) 与if语句结合使用构成循环。其实就是通过if语句额外加了一个循环判断条件。
- (2) 跳出循环体外，特别是多层嵌套退出时使用。

下面通过例子来理解 goto 语句的用法。

### 例程 3-14 goto 语句的用法。

```
#include <stdio.h>
```



```
int main(void)
{
    int i,sum=0;
    i=1;
loop:   if(i<=100)
        {sum=sum+i;
          i++;
          goto loop;}
    printf("%d\n",sum);
    getchar();
    return 0;
}
```

本例程是通过 goto 语句与 if 语句结合构成循环来计算式子  $1+2+\dots+100$  的值。当  $i \leq 100$  时, 就执行语句  $sum=sum+i$ ,  $i++$  以实现累加求和, 再执行 goto loop 实现循环。由于有了循环条件判断  $i \leq 100$ , 当  $i > 100$  时循环就会结束, 不会因为 goto 语句的无条件转移功能而导致死循环。本例程的运行结果为:

5050

### 3.11 循环的嵌套

循环中包含循环就叫做循环的嵌套。在解决实际问题时, 有些问题只用一层循环往往无法解决。因此必须使用多层循环解决, 即循环的嵌套。例如计算式子:  $a+a^2+a^3+\dots+a^n$ 。

三种循环 (for、while、do-while) 可以互相嵌套。可以自由组合成9种嵌套形式。

下面通过例子来理解循环的嵌套使用。

#### 例程 3-15 计算式子 $3+3^2+3^3+\dots+3^{10}$ 的值。

```
#include <stdio.h>
int main(void)
{
    int i,j,t=3;
    long t=3,sum=0;
    for(i=1;i<=10;i++){
        for(j=1;j<i;j++){
            t=t*3;
            sum=sum+t;
            t=3;
        }
    }
    printf("The result is\n%ld",sum);
    getchar();
    return 0;
}
```

这是一个典型的二重循环的例子。对于求解式子  $3+3^2+3^3+\dots+3^{10}$ , 可以分解成两步:

- (1) 利用循环求解出  $3^n$ ,  $n=1, 2, 3, \dots, 10$ 。
- (2) 再利用循环求解出  $3+3^2+3^3+\dots+3^{10}$ 。

因此应用嵌套的 for 语句来计算。内层循环用来求解  $3^n$ ,  $n=1, 2, 3, \dots, 10$ ; 外层循环用来求解出总和  $3+3^2+3^3+\dots+3^{10}$ 。本例程的运算结果为:



```
The result is  
88572
```

## 3.12 break 语句

在前面已经介绍过 break 语句的用法。在 switch 语句中应用 break 语句可以实现多分支选择；在循环语句中应用 break 语句可以实现强制跳出循环。break 语句的一般形式为：

```
break;
```

它的功能就是强制跳出 switch 分支和循环。因此，除了 switch 语句和循环语句，break 语句不能应用在其他任何语句中。

下面再通过例子巩固理解break语句。

### 例程 3-16 break 语句的用法。

```
#include <stdio.h>
int main(void)
{
    char c;
    while(1)
    {
        c=getch();
        printf("%c ", c);
        if(c==27)
            break;
    }
    return 0;
}
```

在本例程中，while 语句的循环判断表达式是 1，也就是如果不强制跳出循环，循环将无休止地进行下去。因此要用 break 语句在适当的时候做强制跳出循环操作。本例程的功能是接收键盘输入的字符，直到按下 Esc 键为止。Esc 键对应字符的 ASCII 码是 27，因此这里用 if 条件语句判断，当键入 Esc 时结束循环。

这里强调一点，所谓用break语句结束循环，就是指程序不再执行本循环语句，而是继续执行循环语句下面的其他语句，在本例程中就是执行return 0语句。它与接下来要讲到的continue语句作用不一样。

## 3.13 continue 语句

continue 语句也是一种强制跳出语句，但作用与 break 不同。它的一般形式是：

```
continue;
```

continue 语句的作用是结束本次循环。即跳过循环体中剩余的语句而强制执行下一次循环。continue 语句只终止本次循环，而并不结束整个循环过程；break 语句则是终止整个循环过程，继续执行循环语句下面的其他语句。

下面通过例子来理解continue语句。



**例程 3-17** continue 语句的用法。

```
#include <stdio.h>
int main(void)
{
    char c;
    while(1)
    {
        c=getch();
        if(c>='0'&&c<='9')
            continue;
        printf("%c ", c);
        if(c==27)
            break;
    }
    return 0;
}
```

本例程在例程 3-16 的基础上又添加了一个功能：在屏幕上不打印出数字字符。也就是说通过本程序只能向屏幕上输出非数字的字符，按 Esc 键时程序结束。这里要用到 continue 语句。当输入的字符为数字字符时，即表达式  $c>='0' \&\& c<='9'$  的值为真时，程序执行 continue 语句，这时程序结束本次循环，也就不再执行 printf 语句，于是数字字符不显示。而当按下 Esc 键时，程序执行 break 语句，这时结束整个循环，于是程序执行 return 0 语句。

### 3.14 本章程序举例

本章重点介绍了C语言程序的三种结构，以及C语言的语法知识。本节将结合具体程序实例来深入理解上面所讲的知识。

**例程 3-18** 有一个分段函数：

$$y = \begin{cases} x & x < 1 \\ 2x-1 & 1 \leq x < 10 \\ 5x+2 & x \geq 10 \end{cases}$$

编写一个程序，输入自变量x，输出函数y。

**分析：**这是一个典型的分支程序设计的例子。根据输入的自变量x的范围不同，选择执行不同的函数表达式。程序设计上，最好选用if语句，因为要根据x的范围选择程序执行的分支，因此条件判断要用关系表达式。而switch语句的条件判断是case后面的常量表达式。所以，要用switch语句实现比较困难。

下面给出程序代码：

```
#include <stdio.h>
int main(void)
{
    float x,y;
    printf("Input x:\n");
    scanf("%f",&x);
    if(x<1)
        y=x;
    else if(x>=1&&x<10)
        y=2*x-1;
```



```
        else
            y=5*x+2;
    printf("y=%f",y);
    getchar();
    return 0;
}
```

给定不同范围上的三个自变量  $x_1=-2.5$ 、 $x_2=5.0$ 、 $x_3=11.2$ ，用该程序执行的结果为：  
当  $x_1=-2.5$  时

```
Input x:
-2.5
y=-2.500000
```

当  $x_2=5.0$  时

```
Input x:
5.0
y=9.000000
```

当  $x_3=11.2$  时

```
Input x:
11.2
y=58.000000
```

**例程 3-19** 输入一个字符串，统计该字符串中的空格符、制表符、回车符的个数。该字符串以 EOF 结束。

分析：因为要连续地输入字符，所以必须用一个循环来实现，又因为该字符串以EOF结束，所以循环条件就是输入的字符不是EOF（Ctrl+Z组合键）。又要分别统计不同类字符的个数，因此要用分支语句实现。这里最好使用switch开关语句，因为它的分支判断只是单个的字符比较，用常量表达式作为判断条件即可。

下面给出程序代码：

```
#include <stdio.h>
int main(void)
{
    char c;
    int tab,space,enter;
    tab=0;
    space=0;
    enter=0;
    do {
        c=getchar();
        switch(c){
            case 9:tab++;break;
            case 32: space++;break;
            case 10: enter++;break;
        }
    }
    while(c!=EOF);
    printf("tab:%d\n",tab);
    printf("space:%d\n",space);
    printf("enter:%d\n",enter);
    getchar();
    return 0;
}
```



本例程是循环语句和分支语句的综合使用。这里用到 do-while 循环语句实现连续地输入字符，直到输入 EOF 为止；对于输入的每个字符，都要通过 switch 语句判断其种类，并根据字符的不同种类在相应的统计变量上加 1 来统计字符的个数。按下 ctrl+Z 组合键，再按回车键，即可出现结果。本例程的运行结果为：

```
This is a test.
      ok!^Z
tab:1
space:3
enter:1
```

### 例程 3-20 打印出“水仙花数”。

所谓“水仙花数”就是指一个3位数，各位数字的立方和等于该数本身。例如：153就是一个水仙花数，因为 $153=1^3+5^3+3^3$ 。

分析：首先水仙花数一定是3位数，因此它的范围就是100~999，那么编程时就是要从这900个数中找到所谓的水仙花数。这显然需要执行一个循环。再对这个范围中的每个数进行判断，如果是水仙花数就打印出来，否则不打印。

判断水仙花数的过程可以这样设计：先将要判断的数的个位、十位、百位提取出来，然后求它们的立方和看是否等于原数，如果等于，则满足水仙花数的条件，于是判断它是水仙花数；如果不等于，则不是水仙花数。

下面给出程序代码：

```
#include <stdio.h>
int main(void)
{
    int i;
    int a,b,c;
    for(i=100;i<=999;i++){
        a=i%100%10;    /*个位数*/
        b=i%100/10;    /*十位数*/
        c=i/100;        /*百位数*/
        if(i==a*a*a+b*b*b+c*c*c)
            printf("%d ",i); /*打印水仙花数*/
    }
    getchar();
    return 0;
}
```

这里用 for 语句实现循环。i 为循环变量，i 的初值为 100，循环条件是  $i \leq 999$ ，这样就将 100~999 这 900 个数都循环到了。再将每个数的个位、十位、百位提取出来，求它们的立方和看是否等于原数。如果判断是水仙花数就打印出来。本例程的运行结果为：

```
153 370 371 407
```

### 例程 3-21 计算式子 $1! + 2! + \dots + 10!$ 的和。

分析：这是一个典型的利用二重循环嵌套求解的问题。求阶乘是一重循环，再累加求和构成二重循环。在程序设计上，内层循环应为求每个数的阶乘，外层循环实现每个阶乘值的累加求和。

下面给出程序代码:

```
#include <stdio.h>
int main(void)
{
    int i,j;
    long t,s=0;
    for(i=1;i<=10;i++){
        for(j=1,t=1;j<=i;j++){
            t=t*j;
            s=s+t;
        }
        printf("The result is:%ld",s);
        getch();
        return 0;
    }
}
```

本程序中,通过内层的 for 语句计算出 1~10 每个数的阶乘,并将阶乘值存入变量 t 中;外层的 for 语句实现每个阶乘值的累加求和,并将求得的结果存放到变量 s 中。最后在屏幕上显示出该结果。本例程的运行结果为:

```
The result is:4037913
```

### 例程 3-22 找出 50~100 内的全部质数。

分析:这仍然是一个典型的利用二重循环嵌套求解的问题。前面已经讲过判断一个数是否是质数的方法,需要用到循环判断,在这里要筛选出 50~100 的所有质数,又要对 50~100 中的每个数进行判断,这也需要一层循环来实现。因此,解决该问题需要用到二重循环。在程序设计上,内层循环的作用是判断该数是否是质数;外层循环的作用是筛选出 50~100 的所有质数。

下面给出程序代码:

```
#include <stdio.h>
int main(void)
{
    int i,j;
    for(i=50;i<=100;i++){
        for(j=2;j<i;j++){
            if(i%j==0) break;
            if(i==j) printf("%d ",i);
        }
        getch();
        return 0;
    }
}
```

本程序内层循环用 for 语句实现,判断 i 是否是质数。若 i 除了 1 和它本身之外还能被其他数整除,则 i 不是质数,于是用 break 语句跳出内层循环;若 i 等于 j,则表示内层循环顺利结束,也就表明 i 是质数。外层循环规定了循环变量 i 的范围。本例程的运行结果为:

```
53 59 61 67 71 73 79 83 89 97
```



## 3.15 本章小结与要点回顾

本章介绍了C语言的基本语句和基本结构。通过对本章的学习，读者可以掌握编写C程序的基本方法，从而能够编写出简单的C程序。下面将本章所介绍的重点知识归纳如下。

### 1. C 语句

C 语句共分为以下 5 类。

- (1) 控制语句：9种控制语句。
- (2) 函数调用语句：例如printf("Hello World!!");语句。
- (3) 表达式语句：由表达式加上分号“;”组成。
- (4) 空语句：由分号“;”组成。
- (5) 复合语句：把多条语句用“{ }”括起来形成一条复合语句。

### 2. C 程序的结构

- (1) 顺序结构：计算机从程序的第一条语句开始执行，直到程序的最后一条语句结束，中间没有“分叉”。
- (2) 分支结构：在程序执行过程中发生“分叉”，根据不同的条件执行不同路径的程序段。
- (3) 循环结构：在程序执行过程中，重复执行相同类型的操作多次。

### 3. 基本的赋值语句

- (1) 赋值运算符用“=”表示，由“=”连接的式子称为赋值表达式。
- (2) 赋值表达式加上分号“;”就构成了赋值语句。
- (3) 如果赋值运算符两边的数据类型不相同，系统会将赋值号右边的类型换成左边的类型。
- (4) 在赋值符“=”之前加上其他二目运算符可构成复合赋值符。例如+=, -=, \*=, /=, %=, <<=, >>=, &=, ^=, |=。

### 4. if 语句

- (1) if语句通过判断给定的条件（一般是关系表达式或逻辑表达式）是否为真，从两组分支操作中选择其中一组。if语句有3种形式。

第一种形式：

```
if(表达式) 语句
```

第二种形式：

```
if(表达式)
    语句 1;
else
    语句 2;
```

第三种形式:

```
if(表达式1)
    语句1;
else if(表达式2)
    语句2;
else if(表达式3)
    语句3;
...
else if(表达式m)
    语句m;
else
    语句n;
```

(2) if 语句的嵌套使用, 有两种形式的嵌套。

```
if(表达式)
    if() 语句1
    else 语句2
```

或者为:

```
if(表达式)
    if() 语句1
    else 语句2
else
    if() 语句1
    else 语句2
```

## 5. switch 语句

switch 语句又叫做开关语句, 常用于多分支选择。它的一般形式为:

```
switch(表达式){
    case 常量表达式1: 语句1;
    case 常量表达式2: 语句2;
    ...
    case 常量表达式n: 语句n;
    default          : 语句n+1;
}
```

一般情况下, switch 语句要与 break 语句结合使用才能实现分支语句的功能。

## 6. for 语句

for 语句是 C 语言中最为灵活的一种循环语句。它的一般形式为:

```
for(表达式1; 表达式2; 表达式3) 语句
```

其中, 语句既可以是简单的语句, 又可以是用“{}”括起来的复杂语句。for 语句的执行过程为:

- (1) 先求解表达式1。
- (2) 求解表达式2, 若其值为真(非0), 则执行for语句中指定的内嵌语句, 然后执行下面第(3)步; 若其值为假(0), 则结束循环, 转到第(5)步。
- (3) 求解表达式3。
- (4) 转回上面第(2)步继续执行。
- (5) 循环结束, 执行for语句下面的语句。



## 7. while 语句

while 语句也是一种应用十分广泛的循环语句。它的一般形式为:

```
while(表达式) 语句
```

执行过程是: 计算表达式的值, 当值为真(非0)时, 执行循环体语句。

## 8. do-while 语句

do-while 语句与 while 语句类似, 只是先执行循环体语句, 后进行循环判断。它的一般形式为:

```
do  
    语句  
while(表达式);
```

执行过程是: 先执行循环体语句, 再进行循环判断, 如果表达式值为真, 则循环继续; 如果表达式值为假, 则循环结束。

## 9. goto 语句

goto 语句又叫做无条件转移语句, 其本质属于转向语句, 但它在程序设计中往往与“标号”构成循环语句, 能起到无条件循环的作用。它的一般形式为:

```
goto 语句标号;
```

执行过程是: 程序执行到这一句, 无条件跳转到语句标号标记的位置继续执行。

## 10. 循环的嵌套使用

(1) 循环中包含循环就叫做循环的嵌套。

(2) 三种循环(for、while、do-while)可以互相嵌套, 自由组合成9种嵌套形式。

## 11. break 语句

break 语句的一般形式为:

```
break;
```

其作用是强制跳出本循环语句, 执行下面的语句。

## 12. continue 语句

continue 语句的一般形式为:

```
continue;
```

其作用是结束本次循环, 即跳过循环体中剩余的语句而强制执行下一次循环。注意与 break 语句作用不同。

本章介绍 C 语言中最重要的概念之一——函数。函数是 C 程序实现的基础，是结构化程序设计的基本单位，是实现特定功能的基本模块。一个完整的 C 程序是通过函数之间的相互调用实现的。因此，掌握函数的知识对学好 C 语言十分关键。

## 4.1 函数概述

函数，就是能够实现特定功能的程序模块。函数是 C 程序的基本功能单位，通过对函数模块的调用实现特定的功能。在 C 程序设计中，要使得程序结构化，就应当将一些具有特定功能的程序段编制成函数，由主函数来调用。这样不仅能使程序简洁、便于调试、可读性强，还可以减少编程的工作量（因为一个函数可以被反复调用）。

C 语言不仅提供了极为丰富的库函数（如 Turbo C，MS C 都提供了三百多个库函数），而且还允许用户建立自己定义的函数。用户在编写程序时，应当充分利用 C 语言提供的这个功能，将能够完成一定功能的程序段编写成一个独立的函数模块，然后用调用的方法来使用函数。

下面先通过两段程序来对比一下，看看使用函数的优点。

### 例程 4-1 两段程序的比较。

不使用函数：

```
#include <stdio.h>
int main(void)
{
    int i,j;
    for(i=50;i<=100;i++) {
        for(j=2;j<i;j++)
            if(i%j==0) break;
        if(i==j) printf("%d ",i);
    }
    getch();
    return 0;
}
```

使用函数：

```
#include <stdio.h>
int main(void)
{
    int i,j;
    for(i=50;i<=100;i++) {
```



```

        if(isPN(i))          /*调用函数语句作为判断条件*/
            printf("%d ",i); /*如果是质数,打印出来*/
    }
    getch();
    return 0;
}
int isPN(int i){
    /*定义函数 isPN(),判断 i 是否是质数
    若 i 是质数,则返回 1; 否则返回 0 */
    int j;
    for(j=2;j<i;j++)
        if(i%j==0) break;
    if(j==i)
        return 1;
    else
        return 0;
}

```

上面两段程序的功能是一样的,都是打印出 50~100 之间的质数。但第一个程序没有使用函数,而是应用了一个二重循环。从表面上看,第一个没有使用函数的程序似乎更为简单,但实际上,程序的可读性、结构都不如使用了函数的程序好。用一个函数 isPN 来判断 i 是否是质数,显然比用一层循环判断更容易理解。另外本例程很简单,只用到一处判断质数的功能。如果在一个较大的程序里要多次判断质数,应用上面的不使用函数的方案设计程序就十分麻烦,不但代码量大,而且代码会十分混乱,最终导致程序无法设计。因此,在程序设计中应当善于使用函数。

一个 C 程序本身就是一个主函数,而在主函数中,又要调用许多具有特定功能的函数来实现整个程序要完成的任务。因此可以讲, C 程序的全部工作都是由各式各样的函数来完成的,故又把 C 语言称为函数式语言。也正是有了函数, C 语言才更易于实现结构化程序设计。

在 C 语言中可从不同的角度对函数分类。

### 1. 从函数定义的角度看,函数可分为库函数和用户自定义函数

(1) 库函数:又叫做标准函数,由系统提供,用户不必定义这些函数就可以直接调用。但调用库函数时,要在程序前包含有该函数原型的头文件。这一点在第 1 章中有所提及。有关 C 语言库函数的知识将在第 2 部分中详细介绍。

(2) 用户自定义函数:由用户自己定义的解决特殊问题的函数。上面例程中的函数 isPN 就是用户自定义函数。

### 2. 从函数结果的角度看,又可将函数分为有返回值函数和无返回值函数

(1) 有返回值函数:函数执行完要返回一个值,该返回值也称为函数的值。上面例程中的函数 isPN 就是有返回值函数,它判断参数 i 是否为质数,如果是质数返回 1;否则返回 0。

(2) 无返回值函数:函数只是执行它的任务,执行完后不返回任何值。由于函数无需返回值,因此用户在定义此类函数时可指定它的返回为“空类型”,空类型的说明符为 void。

### 3. 从函数的数据传递角度看,又可将函数分为有参函数和无参函数

(1) 有参函数:也称为带参函数,即主调函数和被调函数之间有数据的传递。上面例

程中的函数 `isPN` 就是有参函数，其参数是 `i`。也就是说主函数通过调用函数 `isPN` 向函数中传送了 `i`，用来判断 `i` 是否为质数。

(2) 无参函数：主调函数和被调函数之间不进行参数传递。无参函数一般用来执行指定的一组操作。

## 4.2 函数的定义

在程序设计中要使用函数，必须先定义这个函数。前面已经讲过，库函数由系统提供，用户不必定义这些函数就可以直接调用。但是，如果要用到解决特定问题的函数，而库函数中又没有提供，用户就必须自己定义函数。

### 1. 无参函数的定义一般形式

```
类型标识符 函数名()  
{ 声明部分  
  语句  
}
```

其中，类型标识符用来指定函数的返回值类型。对于无返回值类型的函数来说，可以定义为“空类型”，空类型的说明符为 `void`。声明部分是对函数体内部的变量类型说明，语句则是函数要执行的内容。

### 2. 有参函数的定义一般形式

```
类型标识符 函数名(形式参数表列)  
{ 声明部分  
  语句  
}
```

其中，函数名后面的括号里的内容是形式参数表列。它是主调函数和被调函数之间的接口，通过这个形式参数表列，主调函数向被调函数传递数据。其他部分与无参函数一样。结合下面给出的具体函数来理解函数定义的一般形式。

无参函数：

```
void print(){  
printf("Hello World!! ") /*语句*/  
}
```

有参函数：

```
int min(int a,int b){  
int r;      /*声明部分*/  
if(a>=b)    /*语句1*/  
    r=b;  
else  
    r=a;  
return r;   /*语句2*/  
}
```

上面两个函数定义分别为无参函数 `print` 和有参函数 `min`。无参函数 `print` 的功能是在屏幕上输出字符串 `"Hello World!!"`。该函数无返回值，无声明部分，函数体中只有一条语句。



有参函数 min 的功能是得到 a, b 中较小的数并返回。该函数的返回值类型为整型, 变量声明部分为 int r, 有两条语句: if 语句和 return 语句。

对于有返回值类型, 如果在函数定义时不指定它的返回值类型, 则系统默认为返回 int 型。对于参数表列, 凡是定义的有参函数, 一定要在括号内指定参数的类型, 例如 min(int a, int b) 等。因为 C 语言以前的版本中允许在括号外指定参数类型, 而根据 ANSI 新标准, 要求统一在括号内指定参数类型。

最后要强调的一点是, 函数的定义与函数的调用是两个不同的概念。函数的定义是对函数本身的描述, 是函数调用的基础; 函数的调用是在程序执行中使用已定义好的函数, 用它来实现程序的功能。如例程 4-1 中:

```
...  
if(isPN(i))  
...
```

这里的 isPN(i) 是函数的调用, 而函数 isPN 本身的描述是函数的定义。

## 4.3 函数的调用

前面已经简单介绍了函数调用与函数定义的区别, 本节将重点探讨函数调用的一般形式和方法。

### 1. 函数调用的一般形式

函数名(实参表列)

无参函数调用时没有实参表列, 但括号不能省略。实参表列中的参数可以是常数、变量或其他类型的数据及表达式, 各参数之间要用逗号分开。

在函数定义时, 参数表列中的参数叫做形式参数, 简称形参。而在函数的调用中, 参数表列中的参数是实际的参数, 因此叫做实际参数, 简称实参。形参与实参的个数应该相等, 类型应当一致, 名字可以不同。

### 2. 函数调用的方法

在 C 程序设计中, 根据函数在程序中出现的位置不同, 有 3 种函数调用的方法。

#### (1) 函数表达式

函数作为表达式中的一项出现在表达式中。例如:

```
x=min(a,b);  
y=++max(a,b);
```

从例子中不难发现, 函数作为表达式中的一项必须有函数值, 也就是说函数表达式中的函数必须有返回值, 不能是无返回值的函数。本例中函数 min 返回 a, b 中较小的值, 并将该值赋值给 x, 函数 max 返回 a, b 中较大的值, 并将该值自增 1, 然后赋值给 y。

#### (2) 函数作为语句

函数也可以直接作为一条语句出现, 只要在函数调用后面加 “;” 即可。一般情况下,

函数作为语句出现时是没有返回值的，只要求函数完成一定操作。例如：

```
printf("Hello World!!");  
print();
```

其中第一个函数 `printf` 是库函数，在这里实现向屏幕输出字符串 "Hello World!!" 的功能。第二个函数 `print` 是用户自定义函数，同样实现向屏幕输出字符串 "Hello World!!" 的功能，但它是无参函数。

### (3) 函数调用作为参数

函数的参数不但可以是各种类型的数据，还可以是函数的调用。例如：

```
printf("%d",min(a,b));
```

这里函数 `printf` 的一个实参就是函数 `min(a,b)`。一个函数作为另一个函数的参数时必须注意，作为参数的函数一定是调用状态，而且该函数必须有返回值，其返回值作为外层函数的实参。结合本例就是：`min(a,b)` 是函数 `min` 的一次调用，它的返回值作为函数 `printf` 的一个实参。

## 3. 实参的求值顺序

在函数参数表列中，如果参数的个数不止一个，就存在着所谓实参的求值顺序问题。有两种实参的求值顺序：自左至右和自右至左。这要根据不同的系统而定。在 TC 中规定实参的求值顺序是自右至左的。

### 例程 4-2 实参的求值顺序。

```
#include <stdio.h>  
int main(void)  
{  
    int i;  
    i=1;  
    printf("%d,%d,%d",++i,++i,i);  
    getchar();  
    return 0;  
}
```

本例程中，库函数 `printf` 的三个实参 `++i`、`++i`、`i` 的求值顺序是自右至左的。即：先计算 `i`，然后计算 `++i`，最后再计算 `++i`。由于 `i` 的初值是 1，因此，先将 1 传递到函数 `printf` 内部；然后计算 `++i`，此时 `i=2`，把 2 传递到函数 `printf` 内部；最后再计算 `++i`，这时 `i=3`，再把 3 传递到函数 `printf` 内部。最后按照规定的输出格式打印出这三个数。本例程的执行结果是：

```
3,2,1
```

试想，如果实参的求值顺序为自左至右，那么本例程的执行结果就会变为：

```
2,3,3
```

读者可以考虑一下这是为什么。



## 4.4 函数的返回值及类型

前面已经提到,按照有无返回值,又可将函数分为有返回值函数和无返回值函数两种。有的函数是希望得到一个确定的计算结果,像函数作为表达式的一项或者函数的调用作为另一个函数的参数等。而有的函数则只需在程序中完成指定的功能即可,无需得到一个计算结果。

对于有返回值函数,它的返回值是通过 `return` 语句获得的。也就是说,定义一个有返回值的函数,函数中必须包含 `return` 语句。当函数执行到 `return` 语句时,就按照 `return` 关键字后面要求返回的内容把结果返回给主调函数,即使被调函数后面还有语句也不再执行。因此 `return` 语句可以看作是函数执行的结束标志。例如下面一段函数:

```
int fuc(int x){
    if(x>=0)
        return 1;
    else
        return 0;
}
```

它的作用是判断 `x` 的正负,若 `x` 大于等于 0,则函数返回 1;若 `x` 小于 0,则返回 0。函数中有两个 `return` 语句,分别是 `if` 语句的两个分支。当执行完其中任何一个 `return` 语句后,该函数也就执行完毕了。

关于 `return` 语句有几点说明:

(1) `return` 语句后面既可以加括号也可以不加括号。例如:

```
return x;
```

等价于

```
return (x);
```

(2) `return` 语句后面可以跟一个值也可以跟一个表达式。例如:

```
return 1;
return x;
return a+b;
```

都是合法的。

所谓函数值的类型一般就是指函数返回值的类型。一个函数若有返回值,其返回值必定属于一个类型,这个类型也叫做该函数值的类型。在定义函数时,要指定函数值的类型,如果不指定,系统默认函数值类型为整型。例如:

```
int max(int a,int b)
char min(char a,char b)
double aver(double x,double y)
min(int a,int b)
```

以上 4 个函数的函数值分别为:整型、字符型、双精度整型、整型。其中最后一个函数的声明没有指定函数值类型,系统默认函数值类型为整型。

但是有时会出现函数值的类型与返回值的类型不一致的情况。看下面的例子:

```
int min(double x,double y){
    if(x>=y)
        return y;
    else
        return x;
}
```

返回值  $x$ 、 $y$  的类型都是双精度 `double`，而函数值的类型是整型 `int`。这种情况下要以函数值的类型为准。也就是说要将返回值  $x$  或  $y$  先转换为整型，然后以整型形式返回主调函数。下面通过一个完整的实例来理解。

#### 例程 4-3 返回值类型与函数值类型不同。

```
#include <stdio.h>
int main(void)
{
    float x,y;
    float r;
    scanf("%f",&x);
    scanf("%f",&y);
    r=min(x,y);
    printf("%f",r);
    getchar();
    return 0;
}

int min(float x,float y)
{
    if(x>=y)
        return y;
    else
        return x;
}
```

本例程中，函数 `min` 的作用是返回浮点型数  $x$  和  $y$  中较小的一个，然而其函数值类型为整型，这种情况下要以函数值的类型为准，因此将返回值  $x$  或  $y$  先转换为整型，然后再返回主函数。本例程的运行结果为：

```
1.2
2.3
1
```

因为 1.2 小于 2.3，所以应当返回 1.2。但是函数值的类型是整型，因此将 1.2 转换为整型 1 后返回给主函数。

对于无返回值的函数，一般定义为返回一个空类型值，用符号 `void` 说明。如果不指定返回值类型，而且该函数又确实不需要返回值，则系统返回一个不一定有意义的值。例如：

```
void Messenger()
{
    printf("Is error!!");
}
```

就是一个无返回值的函数。



## 4.5 函数的参数及传递方式

前面已经讲过，在函数定义时，参数表列中的参数叫做形式参数，简称形参。而在函数的调用中，参数表列中的参数叫做实际参数，简称实参。一般情况下，人们使用的有参函数要多一些，因此就涉及主调函数和被调函数之间的数据传递问题，也就是实参和形参之间的数据传递问题。

在 C 语言中规定，实参变量和形参变量之间的数据传递是“值传递”，也就是单向地传递。数据只由实参传递给形参，而不能反向传递。当主调函数调用被调函数时，系统为形参变量分配空间，然后将实参的值复制到分配好的形参空间中，以便让它参与函数的运算。因此，实参单元与形参单元是不同的单元，如图 4-1 所示。

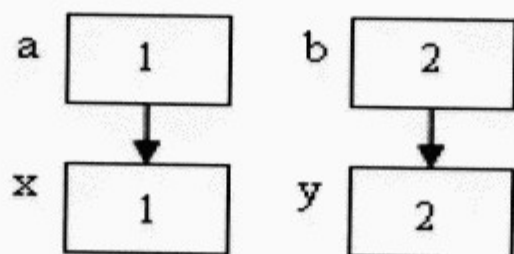


图 4-1 实参与形参之间的数据传递

图 4-1 中 a, b 为实参变量, x, y 为形参变量。当主调函数调用被调函数时, 系统会自动为被调函数分配一段内存空间用来存放函数执行中的变量。因此也就为形参变量 x, y 分配了内存空间, 用来接收实参传递下来的数据。在值传递的过程中, 形参变量 x, y 接收了来自实参变量 a, b 的数据 1 和 2, 这样就完成了实参变量和形参变量之间的数据传递。

当函数执行完毕后, 系统会自动释放掉刚才为被调函数分配的一段内存空间, 因此形参变量 x, y 的内存单元也就会被释放掉。这样无论形参变量 x, y 在函数中发生怎样的变化, 实参变量 a, b 中的数值都不会发生改变。

实际上, 形参变量中的值只是实参变量的一个拷贝, 除此之外二者没有其他任何联系。形参变量只是用来接收实参变量的值, 然后带到函数中参加运算。在执行一个被调函数时, 形参的值即使发生改变, 也不会改变主调函数的实参的值。理解这一点十分重要, 看下面这个例子。

### 例程 4-4 实参和形参之间的数据传递。

```
#include <stdio.h>
int main(void)
{
    int x,y;
    x=1;
    y=2;
    rev(x,y);
    printf("%d,%d",x,y);
    getchar();
    return 0;
}
int rev(int a,int b){
    int t;
```



```
t=a;  
a=b;  
b=t;  
}
```

有些读者看过程序后会认为函数 rev 的功能是将 x, y 的值对调, 如果最初 x 的值为 1, y 的值为 2, 通过 rev 函数的处理后, x 的值变为 2, y 的值变为 1。然而本例程的运行结果为:

1, 2

考虑错误的原因就在于没有真正理解函数之间的值传递。

当主函数调用函数 rev 时, 系统为形参变量 a, b 分配了内存空间用来接收实参变量的数据。由于 x 的值为 1, y 的值为 2, 所以将 1 传递给 a, 将 2 传递给 b。接下来就是执行 rev 函数的操作了, 即将形参变量 a、b 中的值对调, 然而所做的一切与实参 x 和 y 都毫无关系。当函数执行完毕, 系统释放掉函数所用的一切变量, 这样形参变量 a 和 b 也都被释放掉了, 程序重新回到调用 rev 函数那点, 继续执行主函数的其他语句。因此函数 rev 在这里没有起到任何作用, x、y 变量中的值依然不变。

如果想要通过改变形参来改变实参的值, 靠单纯的数值传递是解决不了的, 因为一旦将数值从实参变量传递给形参变量, 二者就不再发生关系, 可以通过传递变量的地址来实现这个功能。这部分知识将在后续章节中介绍。

## 4.6 函数的嵌套调用

在第 4.3 节中已经介绍了函数的调用。其实函数不但可以直接调用(通过主函数调用), 而且可以嵌套调用。在 C 语言中函数不允许嵌套定义, 所以在定义函数时, 函数体内不允许再定义一个函数。但是函数可以嵌套调用, 即在调用一个函数的过程中又调用另一个函数。可以用图 4-2 来形象地描述函数嵌套调用的过程。

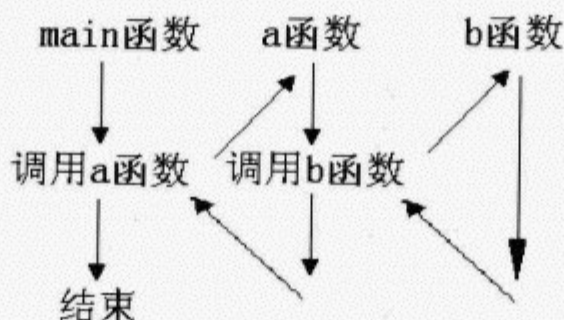


图 4-2 函数的嵌套调用

如图 4-2 所示, 主函数调用 a 函数, a 函数在执行过程中又调用 b 函数, 从而形成了函数的嵌套调用。这里必须注意, 当执行完 b 函数后, 程序回到 a 函数调用 b 函数那一点, 然后继续执行 a 函数的其他语句, 而系统为 b 函数变量分配的内存空间也随即释放。同理, 当执行完 a 函数后, 程序回到主函数调用 a 函数那一点, 然后继续执行主函数的其他语句,



系统为 a 函数变量分配的内存空间也随即释放。

下面通过具体实例来理解函数的嵌套调用。

#### 例程 4-5 计算式子 $1+2^2+3^3+4^4+5^5$ 的值。

分析：该式子可以用二重循环求解，但如果要使程序的结构化和可读性更强，就可以应用函数的嵌套解决。第一层调用的函数是求 n 个值的累加和的函数，该函数的参数是 n；第二层调用的函数是求  $n^n$  的函数。该程序代码如下：

```
#include <stdio.h>
int main(void)
{
    int S;
    S=sum(5);
    printf("The result is:%d",S);
    getchar();
    return 0;
}
int sum(int n){
    int i;
    int s=0;
    for(i=1;i<=n;i++)
        s=s+POW(i);
    return s;
}
int POW(int n)
{
    int i,j=1;
    for(i=1;i<=n;i++)
        j=j*n;
    return j;
}
```

可以看到第一层函数 sum 的参数为 n，它的功能是求基于 1 到 n 的每个数的函数的累加和，用数学表达式可定义为： $\text{sum}(n)=f(1)+f(2)+\dots+f(n)$ ，其中函数 f 即为函数 sum 的嵌套函数，也就是说在函数 sum 中调用了函数 f。对于本题，函数 f 即为函数 POW。函数 POW 的参数是 n，功能是求  $n^n$ 。这样通过函数的嵌套调用就计算出了上式。本例程的运行结果为：

```
The result is: 3413
```

通过本例可以看出，应用函数的嵌套调用，程序的层次性更强、可读性更好，在调试程序时可以逐层函数调试，这样效率会更高。

## 4.7 函数的递归调用

在 C 语言中，不仅允许函数的嵌套调用，还允许函数的递归调用。函数的递归调用是指一个函数在它的函数体内调用它自身。函数的递归调用是函数的嵌套调用的一种特殊形式。在递归调用中，主调函数又是被调函数，这样自己调用自己，一层一层地调用下去。例如函数：

```
int fuc(int a){
```

```

if(a>=100)
return a;
else
    return fuc(++a);
}

```

就是一个典型的递归函数，很显然在函数 fuc 中又调用了函数 fuc。

其实许多数学函数都是用递归的形式定义的。如下所示。

(1) Ackerman 函数：

$$\text{Ack}(m,n)=\begin{cases} n+1 & (m=0) \\ \text{Ack}(m-1,1) & (n=0) \\ \text{Ack}(m-1,\text{Ack}(m,n-1)) & \text{其他情况} \end{cases}$$

(2) Fibonacci 数列：

$$\text{Fib}(n)=\begin{cases} 0 & (n=0) \\ 1 & (n=1) \\ \text{Fib}(n-1)+\text{Fib}(n-2) & \text{其他情况} \end{cases}$$

以上两个数学函数都是用递归的形式定义，因为在每个函数中都调用了其函数本身。

递归是程序设计中的一个强有力的工具。许多问题用其他方法很难解决，而用递归的方法则很容易且代码量很小。特别是针对一些本身就带有递归性质的问题，用递归方法解决更加方便容易。

#### 4.7.1 求 n 的阶乘 n!

下面通过一些实例来理解函数的递归方法。

##### 例程 4-6 求 n 的阶乘 n!。

分析：阶乘问题可以用循环语句轻松解决。但实际上，在数学上阶乘是应用递归形式定义的，n 的阶乘定义为：

$$\text{Fact}(n)=\begin{cases} 1 & (n=0) \\ n \text{ Fact}(n-1) & (n>0) \end{cases}$$

当 n=0 时，n 的阶乘为 1；当 n>0 时，n 的阶乘为 n 乘以 n-1 的阶乘。因此，可以用函数的递归调用来求解它。该程序代码如下：

```

#include <stdio.h>
long Fact(int n){
    if(n==0)
        return 1;
    else
        return n*Fact(n-1);
}
int main(void)
{
    int n;
    long f;
    printf("Please input a number:\n");
}

```



```
scanf("%d",&n);  
f=Fact(n);  
printf("Fact(%d)=%ld",n,f);  
getchar();  
return 0;  
}
```

本程序在主函数中调用函数 Fact 求  $n$  的阶乘。返回一个长整型值赋值给  $f$  作为结果。函数 Fact 的定义同阶乘的数学定义十分相似, 如果  $n$  为 0, 函数返回 1, 否则函数返回  $n * \text{Fact}(n-1)$ 。

在调用函数 Fact 时, 只有当  $n$  的值为 0 时, 函数才停止递归调用而返回一个常量, 否则函数返回  $n * \text{Fact}(n-1)$ , 实际上就是递归调用函数 Fact。因此, 在编写递归函数时应当特别注意, 一定要有结束递归的条件, 否则程序将无限地递归调用下去。本例程的运行结果为:

```
Please input a number:  
10  
Fact(10)=3628800
```

#### 4.7.2 汉诺塔 (Hanoi) 问题

##### 例程 4-7 汉诺塔 (Hanoi) 问题。

一块板上有 3 根针 A、B、C。A 针上套有 64 个大小不等的圆盘, 大的在下, 小的在上, 如图 4-3 所示。要把这 64 个圆盘从 A 针移到 C 针上, 每次只能移动一个圆盘, 移动可以借助 B 针进行。但在任何时候, 任何针上的圆盘都必须保持大盘在下, 小盘在上。求移动的步骤。

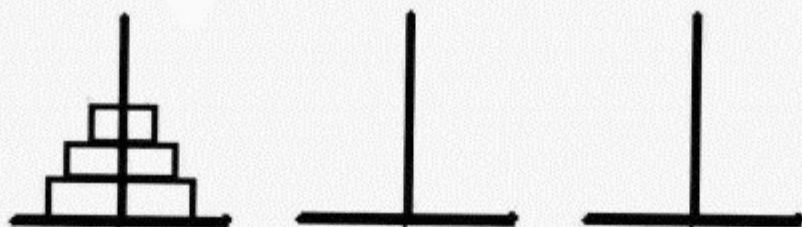


图 4-3 汉诺塔示意图

分析: 这是一个古老的数学问题, 也是一个经典的递归问题。如果要满足上述的要求, 可以这样考虑移动的步骤:

- (1) 先将第 1~63 个盘子移到 B 针上, 要保证大盘在下小盘在上。
- (2) 再将最底下的最大盘子移到 C 针上。
- (3) 最后将 B 针上的 63 个盘子移到 C 针上。

这样问题就解决了。但是关键在于第 (1) 步和第 (3) 步如何执行。由于每次只能移动一个圆盘, 所以在移动的过程中显然要借助另外一根针, 即: 第 (1) 步将第 1~63 个盘子借助 C 针移到 B 针上; 第 (3) 步将 B 针上的 63 个盘子借助 A 针移到 C 针上。这显然又成为两个新的汉诺塔问题了, 即:



问题一：将 A 针上的第 1~63 个盘子借助 C 针移到 B 针上；

问题二：将 B 针上的 63 个盘子借助 A 针移到 C 针上。

解决上述两个问题依然用前面的方法。问题一的圆盘移动步骤为：

- (1) 将 A 针上的第 1~62 个盘子借助 B 针移到 C 针上，要保证大盘在下小盘在上。
- (2) 再将第 63 个盘子移到 B 针上。
- (3) 最后将 C 针上的 62 个盘子借助 A 针移到 B 针上。

问题二的圆盘移动步骤为：

- (1) 将 B 针上的第 1~62 个盘子借助 C 针移到 A 针上，要保证大盘在下小盘在上。
- (2) 再将第 63 个盘子移到 C 针上。
- (3) 最后将 A 针上的 62 个盘子借助 B 针移到 C 针上。

其中第 (1) 步、第 (3) 步的解决又是新的汉诺塔问题，解决的方案与上面一样，直到第 (1) 步和第 (3) 步所移动的盘子个数为 1 时。

在程序的设计上，设函数 `move(int n,int x,int y,int z)`，表示的意思是：将 `n` 个盘子从 `x` 针借助 `y` 针移到 `z` 针上。可以这样定义该函数：

```
move(int n,char x,char y,char z)
{
    if(n==1)
        printf("%c-->%c\n",x,z);
    else
    {
        move(n-1,x,z,y);
        printf("%c-->%c\n",x,z);
        move(n-1,y,x,z);
    }
}
```

递归的结束条件是：`(n==1)`，这时只需要移动一个盘子，无需借助 `y` 针，直接显示移动过程 `x→z`。

否则，递归地调用函数 `move`，按照上面所讲的移动步骤先将 `n-1` 个盘子从 `x` 针借助 `z` 针移到 `y` 针。然后将第 `n` 个盘子直接移到 `z` 针，显示移动过程 `x→z`。

最后将 `y` 上的 `n-1` 个盘子通过 `x` 针移到 `z` 针。

而每次的所谓从“某针借助某针移到某针”都是重新调用 `move` 函数的过程，也就是重新执行上述步骤的过程。直到遇到结束条件 `n==1`，这时直接显示移动过程，并且不再递归地调用 `move` 函数。

下面给出汉诺塔问题的完整解决程序。

```
#include <stdio.h>
move(int n,char x,char y,char z)
{
    if(n==1)
        printf("%c-->%c\n",x,z);
    else
    {
        move(n-1,x,z,y);
        printf("%c-->%c\n",x,z);
        move(n-1,y,x,z);
    }
}
```



```

    }
}
int main(void)
{
    int n;
    printf("input disks number:\n");
    scanf("%d",&n);
    printf("The step to moving %d disks:\n",n);
    move(n,'A','B','C');
    getchar();
    return 0;
}

```

先输入要解决几阶汉诺塔问题，即要移动的盘子数目  $n$ ，然后主函数调用递归函数 `mov`。最开始函数 `mov` 的参数为 `move(n,'A','B','C')`，意思是将 A 针上的  $n$  个盘子通过 B 针移到 C 针，然后执行递归函数 `mov`。在执行过程中 `mov` 函数不断地调用自己。以 3 阶汉诺塔问题为例，最终得到盘子的移动步骤，运行结果如下：

```

input disks number:
3
The step to moving 3 disks:
A-->C
A-->B
C-->B
A-->C
B-->A
B-->C
A-->C

```

通过上面的几个例子可以看出，应用递归方法解决问题有以下特点：问题描述清楚、代码可读性强、结构清晰，代码量一般较非递归方法少。但缺点是递归程序的运行效率比较低，无论是从时间角度还是从空间角度都比非递归程序差。因此对于时间复杂度和空间复杂度要求较高的程序来说，应用函数的递归时要慎重。

## 4.8 局部变量和全局变量

C 语言中所有的变量都有自己的作用域。如果变量声明在主函数中，其作用域就是整个主函数；如果变量声明在被调用的函数中，其作用域就只限于那个被调用的函数。因此，超范围地使用变量是错误的。看下面这个例子。

### 例程 4-8 局部变量的使用 (1)。

```

#include <stdio.h>
int main(void)
{
    int i,j,sum;
    i=1;
    j=2;
    sum=ADD(i,j);
    printf("%d",sum);
    getchar();
    return 0;
}
int ADD(int i,int j){

```

```
sum=i+j;
return sum;
}
```

在主函数中定义了三个整型变量 *i*, *j*, *sum*, 程序目的是实现将 *i*, *j* 相加, 并将结果赋值给 *sum*。这里自定义了一个函数 *ADD*, 用来实现加法的功能。然而, 在函数 *ADD* 的函数体中直接用到在主函数中声明了的变量 *sum*, 企图不再定义新变量而直接使用主函数中定义的, 这是错误的。因为整型变量 *i*, *j*, *sum* 都是在主函数中定义的, 因此, 它们的作用域就是主函数。而在被调用的函数中直接使用主函数中定义的变量, 就属于超范围地使用变量, 因此程序的编译不能通过。

从上面的例子可以看出, 变量说明的方式不同, 其作用域也不同。C 语言中的变量, 按作用域范围可分为两种: 局部变量和全局变量。

### 4.8.1 局部变量

局部变量也叫做内部变量, 它的作用域只限定在该变量所定义的函数内部。也就是说, 局部变量只在本函数内有效, 可以使用, 而在本函数外, 该变量就不能被使用了。上面的例子中, 整型变量 *i*, *j*, *sum* 都属于局部变量, 它们只能在主函数中使用, 离开主函数后再使用这种变量就是非法的。

一般地, 在被调用函数中, 函数的形式参数也是该函数的局部变量。因此, 形参的作用域也只能限定在本函数中。上例中, 函数 *ADD* 的形参 *i*, *j* 都是函数 *ADD* 的局部变量。这里要注意一点, 形参 *i*, *j* 和主函数中定义的整型变量 *i*, *j* 是不同的, 虽然它们同名。主函数中定义的整型变量 *i*, *j* 是主函数的局部变量, 其作用域是主函数; 而形参 *i*, *j* 是函数 *ADD* 的局部变量, 其作用域是函数 *ADD*。在这里给它们起相同的名字只是为了方便, 使代码的可读性增强。因为不同作用域上的变量可以有相同的名字, 而同一作用域上的变量不能重名。

只要将上例中的 *sum* 改成 *ADD* 函数中的局部变量就可以解决错误问题。

#### 例程 4-9 局部变量的使用 (2)。

```
#include <stdio.h>
int main(void)
{
    int i,j,sum;
    i=1;
    j=2;
    sum=ADD(i,j);
    printf("%d",sum);
    getchar();
    return 0;
}
int ADD(int i,int j){
    int sum; /*定义一个局部变量 sum, 注意与主函数中的 sum 不同*/
    sum=i+j;
    return sum;
}
```

这样函数 *ADD* 中的局部变量就包括 *i*, *j*, *sum* 三个, 但要注意, 它们与主函数中定义的 *i*, *j*, *sum* 是不同作用域下的变量, 这里只是重名。本例程的运行结果为:



3

另外在函数内部，可以在复合语句中定义变量，它们只在复合语句中有效，这种复合语句被称为“分程序”或“程序块”。通过下面的例子来理解复合语句中的局部变量。

#### 例程 4-10 复合语句中的局部变量。

```
#include <stdio.h>
int main(void)
{
    int k;
    k=5;
    {
        int k=10;
        printf("k=%d\n",k);
    }
    printf("k=%d\n",k);
    getchar();
    return 0;
}
```

在本程序主函数中定义了一个变量 `k`，并赋值为 5。而在主函数内部包含一个复合语句，也就是分程序。在分程序中定义了局部变量 `k`，并赋值为 10，在屏幕上显示 `k` 的值。分程序执行完毕，再在屏幕上显示主函数中 `k` 的值。由于在分程序中 `k` 是新定义的局部变量，因此在分程序中，`k` 的值为 10；当分程序执行完毕又回到主函数中时，主函数中的 `k` 是局部变量，与分程序中的 `k` 不同，只是重名而已。所以在主函数中 `k` 的值为 5。本例程的运行结果为：

```
k=10
k=5
```

综上，局部变量可以归纳为以下几点：

- (1) 主函数中定义的变量只能在主函数中使用，不能在其他函数中使用。同时，主函数中也不能使用其他函数中定义的变量，因为主函数也是一个函数，与其他函数是平行关系。
- (2) 形参变量是被调函数的局部变量，实参变量是主调函数的局部变量。
- (3) 允许在不同的函数中使用相同的变量名，即不同作用域上的变量可以有相同的名字。它们代表不同的对象，分配不同的单元，互不干扰，也不会发生混淆。
- (4) 在复合语句中也可定义变量，其作用域只在复合语句范围内。

### 4.8.2 全局变量

全局变量又叫做外部变量，与局部变量不同，全局变量是在函数外定义的变量。因此，全局变量不属于任何一个函数，它属于源程序文件。所以，全局变量的作用域也不属于某个函数，而属于整个源程序文件。全局变量可供本文件中的其他函数共同使用，其有效范围是从定义变量的位置开始到源文件的结束。例如：

```
int a,b;           /*全局变量*/
void fuc1()         /*函数 fuc1*/
{
    .....
}
```

```
}
int x,y;          /*全局变量*/
int fuc2()        /*函数 fuc2*/
{
    .....
}
main()           /*主函数*/
{
    .....
}
```

其中变量  $a$ ,  $b$ ,  $x$ ,  $y$  都是全局变量, 因为它们没有定义在函数内部, 不属于任何一个函数, 但是它们的作用域不同。全局变量的作用域是从定义变量的位置开始到源文件的结束。因此, 全局变量  $a$ ,  $b$  的有效范围是函数  $fuc1$ 、 $fuc2$  和主函数  $main$ , 而全局变量  $x$ ,  $y$  的有效范围只是函数  $fuc2$  和主函数  $main$ 。这样对于变量  $a$ ,  $b$ , 函数  $fuc1$ ,  $fuc2$  和主函数  $main$  都可以直接使用, 不需要在函数内部再作说明, 同理对于变量  $x$ 、 $y$ , 函数  $fuc2$  和主函数  $main$  也可以直接使用, 而函数  $fuc1$  则不可以直接使用。

全局变量是一个源程序文件中各个函数共享的“公共资源”。每个函数对全局变量的修改都会影响到全局变量的取值。这就好比教室里的黑板, 它是全班学生的公共资源, 不属于哪一位同学, 因此只要有一位同学在黑板上写字, 全班同学都会看到。而局部变量就像个人的笔记本, 只供拥有者使用。这样, 全局变量就成为了各个函数之间信息互通的纽带, 只要一个函数向全局变量提供数据, 其他函数都可共享。

下面通过具体实例来理解全局变量的用途。

#### 例程 4-11 全局变量的应用。

```
#include <stdio.h>
#define PI 3.14
float C;
float S;
void fuc(float r)
{
    C=2*r*PI;
    S=PI*r*r;
}

int main(void)
{
    float r;
    printf("Please input the r of the circle\n");
    scanf("%f",&r);
    fuc(r);
    printf("C=%f\n",C);
    printf("S=%f\n",S);
    getchar();
    return 0;
}
```

本例程用来计算圆的周长和面积。将存放周长和面积结果的变量  $C$  和  $S$  设为全局变量, 这样无论在主函数中还是在被调函数中都可以使用全局变量。在 C 语言中, 函数只能返回一个确定的值或者没有返回值, 因此要得到周长和面积两个结果必须要两个函数分别来实现。但是, 如果像本例程这样将存放周长和面积结果的变量  $C$  和  $S$  设为全局变量, 问题就简单了。只需定义一个无返回值的函数  $fuc$ , 在该函数内部直接



计算出给定半径的圆的周长和面积，并将结果赋值给全局变量 C 和 S。而作为全局变量，主函数也可以直接使用，因此函数就省掉了返回值，直接从全局变量中获得数据。这样，全局变量就实现了主函数 main 和被调函数 fuc 之间的数据传递和数据共享。本例程的运行结果为：

```
Please input the r of the circle
2.5
C=15.700000
S=19.625000
```

在使用全局变量时，还有一个全局变量与局部变量重名的问题。如果在同一个源文件中全局变量与局部变量重名，则在局部变量的作用范围内，全局变量被“屏蔽”掉，也就是它不起作用。通过例子来理解这一点。

#### 例程 4-12 全局变量的“屏蔽”。

```
#include <stdio.h>
int a=1,b=2;
int ADD(int a,int b)
{
    int c;
    c=a+b;
    return c;
}
void print()
{
    printf("%d,%d\n",a,b);
}
int main(void)
{
    int a,b;
    a=3;
    b=4;
    printf("%d\n",ADD(a,b));
    print();
    getchar();
    return 0;
}
```

本例程中定义 a, b 为全局变量，并赋初值为 a=1, b=2。然而在主函数中又定义了两个局部变量 a, b，产生了重名现象。根据规则，全局变量要被“屏蔽”掉，也就是说，在主函数中被调函数 ADD 的实参为局部变量的值 a=3, b=4，而不是全局变量的值 a=1, b=2。而在被调函数 ADD 中，形参 a, b 也是局部变量，因此它们直接接收实参传过来的值，全局变量 a, b 在函数 ADD 中也不起作用。而函数 print 中没有定义局部变量，因此函数 print 中的 a, b 是全局变量，其作用是将全局变量 a, b 的值显示在屏幕上。本例程的运行结果为：

```
7
1,2
```

使用全局变量可以使源程序文件中的各个函数共享一个公共资源，从而方便函数间的数据传递。特别是有时可以省去函数的返回值，使程序设计更加容易。但过多使用全局变量会破坏整个程序的结构，降低程序的清晰性，导致程序可读性差，程序易出错，而且破

坏了程序设计中最低访问权限的原则，所以不要过度使用全局变量。

## 4.9 变量的存储类别

上节是按照变量的作用域将变量划分为全局变量和局部变量。其实变量的划分方式很多，本节将按照变量的存储方式对变量进行分类。

### 4.9.1 动态存储变量和静态存储变量

从变量的作用域角度划分，可将变量分成全局变量和局部变量两类。如果从变量的生存期角度划分，又可将变量分为动态存储变量和静态存储变量两种。

顾名思义，动态存储变量就是指该变量的存储形式是动态的，也就是说在程序的运行过程中，动态地为该变量分配内存空间；而静态存储变量就是指该变量的存储形式是静态的，也就是说该变量在程序运行期间其内存空间是固定分配的，不能随时分配或者随时撤销。

要想理解动态存储方式和静态存储方式，先要理解内存中用户存储空间的基本情况。在内存中，系统提供给用户的存储空间可分为三个部分：

- ✧ 程序区。
- ✧ 静态存储区。
- ✧ 动态存储区。

其中，程序区用来存放用户要执行的程序段。数据分别存放在静态存储区和动态存储区中。各存储区存放的数据内容如下。

#### 1. 静态存储区

全局变量。在程序的执行过程中，全局变量占据固定的内存空间，直到程序执行完毕才释放掉。

#### 2. 动态存储区

- ✧ 函数形参。只有在调用该函数时才为形参分配内存空间，调用完成后要释放掉所分配的内存空间。
- ✧ 自动变量（未加 `static` 声明的局部变量）。在函数调用时为其分配内存空间，调用完成后要释放掉所分配的内存空间。
- ✧ 函数调用时的现场保护和返回地址。在函数调用时为其分配内存空间存放程序的地址。

以上就是静态存储区和动态存储区中的数据内容。变量根据其生存期要求的不同，分配到静态存储区或动态存储区中。在C语言中，变量的存储方式分为两大类：静态存储类和动态存储类。它们又可包含4种形式的变量：自动（`auto`）、静态（`static`）、寄存器（`register`）和外部（`extern`）。下面做具体的介绍。



### 4.9.2 auto 变量

如果不特意声明为 `static` 存储类别，函数中的局部变量都是动态分配存储空间，数据存储在动态存储区中。这类变量在调用函数时，由系统自动为它们分配内存空间，当函数调用结束时，系统会自动释放这些空间。所以，这类局部变量也称为自动变量。

通常自动变量用 `auto` 关键字作为存储类别声明。例如：

```
void fuc()
{
    auto int x,y;
    ... ..
}
```

其中，局部变量 `x`, `y` 定义为自动变量。当函数 `fuc` 被调用时，系统自动为变量 `x`, `y` 分配内存空间，当函数调用结束时，系统会自动释放这些空间。当然，关键字 `auto` 完全可以省略掉不写。C 语言中规定，`auto` 不写则隐含定为“自动存储类别”，它属于动态存储方式。因此，上面的程序例子等价于

```
void fuc()
{
    int x,y;
    ... ..
}
```

在前面提到的许多例程中，函数的局部变量都是不加关键字 `auto` 定义的自动变量，也就是说当函数调用结束时，系统会自动释放这些变量的存储空间。

### 4.9.3 用 `static` 声明的局部变量

如果在定义局部变量时，在变量的前面加上关键字 `static`，该局部变量就被定义为 `static` 局部变量，即静态局部变量。静态局部变量与自动变量不同，当函数调用结束时，它不会像自动变量那样被系统释放掉，而是保留原值，也就是说静态局部变量所占有的内存空间不因函数调用的结束而被释放。下面通过例子来理解静态局部变量与自动局部变量的区别。

#### 例程 4-13 静态局部变量与自动局部变量的比较。

##### (1) 利用静态变量计算阶乘

```
#include <stdio.h>
int main(void)
{
    int i,n;
    for(i=1;i<=10;i++)
        n=fac(i);
    printf("The result is:%d\n",n);
    getchar();
    return 0;
}
int fac(int i)
{
    static n=1;
    n=n*i;
    return n;
}
```

## (2) 将 n 改为自动变量后

```
#include <stdio.h>
int main(void)
{
    int i,n;
    for(i=1;i<=10;i++)
        n=fac(i);
    printf("The result is:%d\n",n);
    getchar();
    return 0;
}
int fac(int i)
{
    auto int n=1;
    n=n*i;
    return n;
}
```

看上面两段程序。第一段程序是利用静态变量计算阶乘。定义 n 为静态变量，当第一次调用函数 fac 时，给 n 赋值为 1，然后计算 n 与形参 i 的乘积，并把计算结果赋值给 n 返回。这样，当函数 fac 调用结束时，静态变量 n 的内存空间不释放，保持原值。当下一次再调用函数 fac 时，静态变量 n 的值依然保持原值，即为上一次计算的 n 与形参 i 的乘积。如此循环累乘，最终计算出 10 的阶乘。第一段程序的运行结果为：

```
The result is:24320
```

这里需要注意的一点是，对静态变量只赋初值一次，第二次再调用该函数时，虽然有语句 static n=1；但并不赋值，静态变量 n 保持原值。

第二段程序为将 n 改为自动变量后的情况。定义 n 为自动变量，当第一次调用函数 fac 时，给 n 赋值为 1，然后计算 n 与形参 i 的乘积，并把计算结果赋值给 n 返回。而当函数 fac 调用结束后，变量 n 的内存空间被释放掉。这样当程序再次调用函数 fac 时，系统重新为变量 n 分配空间，重新为 n 赋值为 1，于是就起不到循环累乘的作用，当然无法得到正确的结果。第二段程序的运行结果为：

```
The result is:10
```

由于每次都要重新为 n 赋值为 1，所以只返回 i 值本身。

通过上面两段程序的比较可以看出静态变量与自动变量本质上是不同的。前者当函数调用结束时，变量本身的物理空间不释放；后者当函数调用结束时，变量空间被释放掉。不同的原因是它们的存储方式不一样，前者是静态存储方式，在静态存储区内分配存储单元，在程序整个运行期间都不释放；后者则是动态存储方式，占动态存储空间，函数调用结束后即释放。

另外，对于静态变量还要注意以下三点：

(1) 静态局部变量在编译时赋初值，也就是只赋初值一次；而对自动变量赋初值是在函数调用时进行，每调用一次函数重新赋一次初值，相当于执行一次赋值语句。

(2) 如果在定义局部变量时不赋初值，静态局部变量编译时会自动赋初值 0（对数值型变量）或空字符（对字符型变量）；自动变量的值则是一个不确定的值。

(3) 要区分静态局部变量与全局变量。它们的生命周期都是整个程序的执行过程，但作用



域不同。全局变量的作用域是整个源文件，而静态局部变量的作用域只限于它所定义的函数。

#### 4.9.4 register 变量

一般情况下，变量的值都存放在内存中。每次处理数据时，CPU 发出读取数据命令，并从内存传送到 CPU 中的运算器中进行运算。但是，有时有些数据要频繁进行运算，例如循环语句中参与运算的数据等。如果每进行一次运算都将数据存回内存中的话，那么再次读取时也都要从内存中读取，这样每次存取数据花费的时间就过长，导致整个程序的执行效率也就不高。

为了提高效率，C 语言允许将局部变量的值存放在 CPU 中的寄存器里，这种变量叫“寄存器变量”，用关键字 `register` 作声明。这样每次存取数据时，CPU 就可以直接从寄存器中存取数据，而不必通过内存传送。由于从寄存器中读取数据要比从内存中读取数据快得多，因此将一些读取频繁的变量定义为寄存器变量会提高程序的执行效率。

下面通过例子来理解 `register` 变量。

##### 例程 4-14 计算 5 的阶乘。

```
#include <stdio.h>
int fac(int n)
{
    register int i, f=1;
    for(i=1; i<=n; i++)
        f=f*i;
    return(f);
}
int main()
{
    int a;
    a=fac(5);
    printf("5!=%d", a);
    getch();
    return 0;
}
```

本例程是计算 5 的阶乘。在定义计算阶乘的函数 `fac` 时，将循环变量 `i` 和用于存放结果的变量 `f` 定义为 `register` 变量，即寄存器变量。这是因为这两个变量处于循环语句中（在这里要循环 5 次），所以可将变量定义为寄存器变量，以减少 CPU 与内存之间的数据交换，提高程序的执行效率。本例程的运行结果为：

```
5!=120
```

应用寄存器变量可以避免过多地读取内存，从而提高程序的执行效率。但在定义寄存器变量时也要注意以下几点：

- (1) 只有局部自动变量和形式参数可以作为寄存器变量。
- (2) 一个计算机系统中的寄存器数目有限，不能定义任意多个寄存器变量。
- (3) 局部静态变量不能定义为寄存器变量。

#### 4.9.5 同一文件中用 `extern` 声明外部变量

前面已经讲到，外部变量（全局变量）是在函数外定义的变量，它不属于任何一个函数，

源文件中的任何一个函数都可以使用全局变量。全局变量的内存空间分配在静态存储区中。

全局变量的作用域是从全局变量的定义点到源文件的结尾。一般情况下，在全局变量的定义点之前引用该全局变量是非法的。例如：

```
int main(void)
{
    printf("%d",A);
    getchar();
    return 0;
}
int A=1;
```

这种在主函数中直接引用全局变量的做法是非法的，因为全局变量的定义在引用之后，它的作用域是从全局变量的定义点到源文件的结尾，而不包括定义点以上的区域。

但是，C语言中规定可以用关键字 `extern` 来声明全局变量，也就是外部变量，以扩展全局变量的作用域。

上面的例子是非法引用全局变量，但是如果在引用全局变量之前用关键字 `extern` 加以声明的话，这种引用就是合法的。上面的例子就可以修改成：

```
int main(void)
{
    extern int A;
    printf("%d",A);
    getchar();
    return 0;
}
int A=1;
```

这样引用全局变量就是合法的了。用关键字 `extern` 对提前引用的全局变量加以声明的方法叫做外部变量声明。在同一个文件中使用外部变量声明，可以扩大全局变量的作用域，使得在定义全局变量之前就可以引用全局变量。在声明外部变量时，可以像上例那样应用 `extern int A` 语句，也可省略掉类型声明 `int`，直接写成 `extern A`。

#### 4.9.6 多个文件中用 `extern` 声明外部变量

另外，如果不同的文件间要引用同一个外部变量，在不同的文件中各自定义一个同名的外部变量的做法是不允许的。

例如文件 `f1.c` 中内容为：

```
#include <stdio.h>
int A=1;
int main(void)
{
    printf("%d",A);
    print();
    getchar();
    return 0;
}
```

文件 `f2.c` 中内容为：

```
int A=1;
void print()
{
```



```
printf("%d",A);
}
```

在源文件 f1.c 和源文件 f2.c 中都定义了外部变量 A。编译时,将 f1.c 与 f2.c 连接到一起进行编译,主函数 main 自然可以调用函数 print。然而,在源文件 f1.c 和源文件 f2.c 中都定义了外部变量 A,企图将一个外部变量 A 共享于两个文件,这是不允许的。因为它会在程序的编译链接时出现全局变量“重复定义”的错误,即同一个变量不能在一个文件中定义两次。

上面的例子在不同的文件中定义两次全局变量的做法是错误的,因为这是一种重复定义,在程序的编译链接过程中,系统会提出错误。解决此问题的方法是在其中任意一个文件中定义全局变量,然后在另外的文件中用关键字 `extern` 对要引用的全局变量作外部变量声明,这样在不同的文件中就可以引用同一个文件里定义的外部变量了。上面的例子就可以修改如下。

文件 f1.c 中内容为:

```
#include <stdio.h>
int A=1;
int main(void)
{
    printf("%d",A);
    print();
    getchar();
    return 0;
}
```

文件 f2.c 中内容为:

```
extern int A;
void print()
{
    printf("%d",A);
}
```

可以看到在文件 f2.c 中,用关键字 `extern` 对外部变量 A 加以外部变量声明,这样,即使该变量在文件 f2.c 中没有定义而仅在 f1.c 中定义,程序也可以合法引用。两个文件 f1.c、f2.c 共享的外部变量的内存空间分配在静态存储区,使用关键字 `extern` 在多文件中对外部变量加以声明,扩大了全局变量的作用域。

下面通过例子来理解用关键字 `extern` 声明外部变量。

#### 例程 4-15 多文件程序中声明外部变量。

文件 4-15-1.c 中的内容:

```
#include <stdio.h>
#include "4-15-2.c"
#define PI 3.14                /*常量的定义*/
float C;                      /*外部变量定义*/
float S;                      /*外部变量定义*/
int main(void)
{
    float r;
    printf("Please input the r of the circle\n");
    scanf("%f",&r);
    fuc(r);
    printf("C=%f\n",C);
    printf("S=%f\n",S);
    getchar();
}
```

```
    return 0;
}
```

文件 4-15-2.c 中的内容:

```
#define PI 3.14          /*常量的定义*/
extern float C,S;        /*声明外部变量*/
void fuc(float r)
{
    C=2*r*PI;            /*引用外部变量 C*/
    S=PI*r*r;            /*引用外部变量 S*/
}
```

在文件 4-15-2.c 中通过关键字 `extern` 声明文件 4-15-1.c 中定义的外部变量 `C` 和 `S`，文件 4-15-2.c 中的程序也可以引用它们。实际上，两文件共享外部变量 `C` 和 `S`。

之所以要把一个源程序文件拆分成若干个子文件，然后互相包含，最后在编译时合成一个目标文件，就是因为这种做法有利于大型程序系统的开发，可以把繁重的任务按功能细化。因此，当一个文件要共享其他文件中定义的外部变量时，就要用到关键字 `extern` 来声明外部变量。

本例程的运行结果是:

```
Please input the r of the circle
3
C=18.840000
S=28.260000
```

使用声明外部变量的方法共享其他文件中定义的外部变量时要特别慎重，因为一个变量可能被几个文件所共享，只要其中一个文件中的程序对外部变量进行了修改，就可能影响到其他文件对外部变量的使用，从而影响到整个程序的结果。

#### 4.9.7 用 `static` 声明外部变量

前面介绍过用 `static` 声明的局部变量内存分配在静态存储区中，变量的内存空间不随着函数的调用结束而被释放，用 `static` 声明的外部变量与用 `static` 声明的局部变量不是一个概念。

在上一节中讲过，不同的文件可以通过外部变量声明共享一个文件中定义的外部变量（全局变量）。然而有时并不希望一个文件中的外部变量被其他文件共享，这时就可以用关键字 `static` 对定义的外部变量加以声明，这样文件中定义的外部变量就只限于在本文件中引用，而不能被其他文件引用。

例如文件 `f1.c` 中内容为:

```
#include <stdio.h>
static int A=1;
int main(void)
{
    printf("%d",A);
    print();
    getchar();
    return 0;
}
```

文件 `f2.c` 中内容为:

```
extern int A;
```



```
void print()
{
    printf("%d",A);
}
```

在文件 f1.c 中定义了外部变量 A，并且用关键字 `static` 加以声明，这样外部变量 A 只限于在文件 f1.c 中使用，其他文件不能引用。文件 f2.c 企图用 `extern` 进行外部变量声明引用全局变量 A 的做法是无效的。

## 4.10 内部函数和外部函数

C 语言中大部分函数都可以被其他函数调用，包括本文件中的函数和其他文件中的函数。但也有一类函数只能被本文件中的其他函数调用，而不能被其他文件调用，这类函数称为内部函数。相应地，那些也可以被其他文件调用的函数称为外部函数。

### 4.10.1 内部函数

如果一个函数只能被本文件中的其他函数所调用，而不能被其他文件中的函数调用，该函数就称为内部函数。内部函数定义的一般形式为：

```
static 类型标识符 函数名 (形参表列)
```

也就是在通常定义的函数前面加一个 `static`，例如：

```
static int fac(int x)
```

就是一个内部函数的定义，函数名为 `fac`，函数的返回值类型为整型。

定义内部函数有利于大型程序的开发，因为在开发大型程序时，往往将不同功能的函数分配到不同文件中，由不同的程序员编写。如果两个程序员在不同的文件中编写了相同名称的函数，待程序编译时，就会发生函数重定义的错误。如果程序员在定义本文件中的函数时将函数声明为内部函数，可以使函数只局限于所在的文件，使它不与其他文件中的函数发生干扰，这样程序员就不必担心自己所编写的函数是否与其他文件中的函数重名了。

### 4.10.2 外部函数

如果一个函数既能被本文件中的其他函数调用，又能被其他文件中的函数调用，该函数就称为外部函数。外部函数定义的一般形式为：

```
extern 类型标识符 函数名 (形参表列)
```

C 语言中规定，如果在定义函数时省略掉关键字 `extern`，则隐含默认定义的函数为外部函数，因此前面例程中定义的函数都可称为外部函数。例如：

```
extern int fac(int x)
```

就是一个外部函数定义，其他文件中的函数也可以调用该函数。它等价于：

```
int fac(int x)
```

另外，调用外部函数时，在该函数的文件中要用 `extern` 声明所调用的函数为外部函数，

这与声明外部变量的方法类似。

例如文件 f1.c 中的内容为：

```
int main()
{
    ... ..
    extern fuc(int a); /*先用 extern 声明所要调用的外部函数*/
    ... ..
    fuc(a);           /*调用外部函数*/
}
```

文件 f2.c 中的内容为：

```
void fuc(int a)
{ /*在文件 f2.c 中定义外部函数 fuc*/
    ... ..
}
```

在文件 f1.c 中，主函数要调用文件 f2.c 中定义的外部函数 fuc，在正式调用函数 fuc 之前，先用 extern 声明所调用的函数为外部函数。

## 4.11 本章小结与要点回顾

本章主要介绍了 C 语言中函数的基本知识，其中包括函数概述、函数的定义、函数的调用、函数的返回值、函数的参数、函数的嵌套与递归、局部变量与全局变量、变量的存储类别以及内部函数与外部函数。下面归纳一下本章所讲的内容。

### 1. 函数概述

函数是能够实现特定功能的程序模块。

函数是 C 程序的基本功能单位，通过对函数模块的调用实现特定的功能。

从函数定义的角度看，函数可分为库函数和用户定义函数两种。

C 语言的函数兼有其他语言中的函数和过程两种功能，因此，又可把函数分为有返回值函数和无返回值函数。

从函数的数据传送角度来看，又可将函数分为有参函数和无参函数。

### 2. 函数的定义

函数的定义分为无参函数的定义和有参函数的定义两种。

(1) 无参函数定义的一般形式为：

```
类型标识符 函数名()
{ 声明部分
  语句
}
```

(2) 有参函数定义的一般形式为：

```
类型标识符 函数名(形式参数表列)
{ 声明部分
  语句
}
```



```
}
```

### 3. 函数的调用

函数调用的一般形式为：

函数名(实参表列)

另外有三种函数调用的方法。

(1) 函数表达式，例如：

```
x=min(a,b);
```

(2) 函数作为语句，例如：

```
printf("Hello World!!");
```

(3) 函数调用作为参数，例如：

```
printf("%d",min(a,b));
```

有两种实参的求值顺序：自左至右和自右至左。这要根据不同的系统而定，TC 的实参的求值顺序为自右至左。

### 4. 函数的返回值

有的函数希望得到一个明确的结果，所以要有一个返回值；而有的函数则只需要在程序中完成指定的功能即可，所以不需要返回值。

对于有返回值函数，它的返回值是通过 `return` 语句获得的。

对于无返回值的函数，一般定义为返回一个空类型值，用符号 `void` 说明。如果不指定返回值类型，而且该函数又确实不需要返回值，则系统返回一个不一定有意义的值。

### 5. 函数的参数

在函数定义时，参数表列中的参数叫做形式参数，简称形参。

在函数的调用中，参数表列中的参数叫做实际参数，简称实参。

在 C 语言中规定，实参变量和形参变量之间的数据传递是“值传递”，也就是单向地传递。实参将数据传递给形参，形参的一切改变都不影响实参的值。

### 6. 函数的嵌套调用

在 C 语言中函数不允许嵌套定义，但是函数可以嵌套调用。

所谓函数的嵌套调用就是调用一个函数的过程中又调用另一个函数。

应用函数的嵌套调用可以使程序层次性更强、可读性更好，在调试程序时可以对函数逐层调试，使得效率更高。

### 7. 函数的递归调用

函数的递归调用是指一个函数在它的函数体内调用它自身。

应用递归方法解决问题的优点是问题描述清楚、代码可读性强、结构清晰且代码量一般较非递归方法少。

递归方法的缺点是递归程序的运行效率比较低，无论是从时间角度还是从空间角度都比非递归程序差。

## 8. 局部变量与全局变量

局部变量也叫做内部变量，它的作用域只限定在该变量所定义的函数内部。

全局变量是在函数外定义的变量，它不属于任何一个函数，只属于源程序文件。因此，全局变量的作用域属于整个源程序文件。其有效范围是从定义变量的开始位置到源文件的结束。

过多使用全局变量会破坏整个程序的结构、降低程序的清晰性，导致程序可读性差、程序易出错，所以要适度使用全局变量。

## 9. 变量的存储类别

变量的存储方式分为两大类：静态存储和动态存储。

静态存储方式存储的变量包括：

- ✧ 全局变量（外部变量）。
- ✧ 用 `static` 声明的局部变量。

动态存储方式存储的变量包括：

- ✧ 函数形式参数。
- ✧ 自动变量（`auto` 变量）。
- ✧ 函数调用时的现场保护和返回地址。

此外为了使存取更加快捷，程序的执行效率更高，还有寄存器变量（`register` 变量）。

为了在全局变量的定义之前引用该全局变量或在多文件中共享同一个全局变量，可用关键字 `extern` 声明外部变量。

如果不希望文件中的外部变量被其他文件共享，可以用关键字 `static` 对定义的外部变量加以声明，这样它就只限于在本文件中引用，而不能被其他文件引用。

## 10. 内部函数与外部函数

一个函数只能被本文件中的其他函数所调用，而不能被其他文件中的函数调用，这样的函数称为内部函数。

（1）内部函数定义的一般形式为：

```
static 类型标识符 函数名(形参表列)
```

内部函数也称为静态函数。

一个函数既能被本文件中的其他函数调用，又能被其他文件中的函数调用，该函数就称为外部函数。

（2）外部函数定义的一般形式为：

```
Extern 类型标识符 函数名(形参表列)
```



# 预处理命令

本节将详细介绍 C 预处理命令。预处理发生在程序的编译之前，它不属于 C 语言的一部分，但有着十分重要的作用。预处理命令都以“#”开头，处于源程序的最开始，因此预处理命令前面只能出现空白。

## 5.1 预处理命令概述

预处理命令（preprocessor directives）是 ANSI C 标准规定的一种 C 程序控制命令。我们对于预处理命令其实并不陌生，前面多次提到的库文件包含、常量定义都属于预处理命令。

预处理，顾名思义就是在正式处理之前预先进行的处理。所以，预处理命令并不属于 C 语言本身的组成部分。通常所讲的编译、链接也都是针对 C 语言程序本身的操作，对预处理命令的处理不包含在其中，而是在编译、链接操作之前。

预处理是 C 语言的一个重要功能，预处理工作主要由预处理程序负责完成。当要对一个源文件进行编译时，系统先要自动引用预处理程序对源程序中的预处理命令做相应的处理，处理完毕后自动进入对源程序的编译。

下面通过一个实例理解预处理的过程。

### 例程 5-1 程序的预处理过程。

```
#include <stdio.h>
#define PI 3.1416
int main(void)
{
    double r;
    r=1.2;
    printf("The length of the circle is:%f",PI*2*r);
    getchar();
    return 0;
}
```

前面已经提到库文件包含、字符常量定义都属于预处理命令。在本例程中 `#include <stdio.h>` 是文件包含，`#define PI 3.1416` 是常量定义，都属于预处理命令。在正式对本段程序编译之前，预处理程序根据预处理命令的要求，用库文件 `stdio.h` 中的实际内容替代文件包含命令 `#include <stdio.h>`，将字符常量 `PI` 都替换成 `3.1416`。这样，经过预处理后的源程序不再包含预处理命令，而只是单纯的 C 程序，再通过程序的编译、链接最终生成可执行

的目标程序。

以上简单概述了一下程序的预处理过程，C 提供了多种预处理功能，本章主要讨论以下 3 种：

- ✧ 宏定义。
- ✧ 文件包含。
- ✧ 条件编译。

在编程中合理地使用预处理功能，会使程序更加便于阅读、修改、移植和调试，也有利于模块化程序设计。

## 5.2 宏定义及其分类

在 C 程序中，用一个标识符来表示一个字符串，叫做“宏”，这个标识符称为“宏名”。预处理时，源程序中出现的宏名都要被替换为宏中定义的字符串，这个过程称为“宏代换”或“宏展开”。例如上节中提到的将 PI 替换为 3.1415926 的过程就叫做“宏代换”或“宏展开”，其中标识符 PI 称为宏名，3.1415926 是用 PI 表示的字符串，整个的常量定义 `#define PI 3.1416` 叫做一个宏定义。

C 语言中的宏定义分为带参数的宏定义和不带参数的宏定义两种。像前面提到的字符常量定义就属于不带参数的宏定义。而带参数的宏定义有时可代替函数使用，比调用函数效率更高，接下来将分别介绍这两种形式的宏定义。

## 5.3 不带参数的宏定义

不带参数的宏定义的使用非常频繁，最为常见的就是字符常量的定义。另外，不带参数的宏定义还有其他形式和用途。本节将详细介绍不带参数的宏定义。

### 5.3.1 不带参数的宏定义的一般形式

例程 5-1 中提到的这种用一个指定的标识符来表示一个字符串，而没有参数形式的宏定义就是不带参数的宏定义。它的一般形式为：

```
#define 标识符 字符串
```

其中的“#”表示这是一条预处理命令，它不属于 C 语言本身的组成部分，在程序编译之前要做预处理。`define` 为宏定义命令，在做预处理时遇到 `define`，就要做相应的宏代换。“标识符”为所定义的宏名，“字符串”可以是常数、表达式、格式串等。

下面不带参数的宏定义都是合法的：

```
#define PI 3.1416  
#define A (x+y+z)  
#define C PI*R*2
```



对于字符串是表达式的宏定义,像`#define A (x+y+z)`,其作用是指定标识符A来代替表达式 $(x+y+z)$ 。在对源程序编译之前,先由预处理程序进行宏代换,即用 $(x+y+z)$ 表达式去置换所有的宏名A,然后再进行编译。

下面举一个字符串是表达式的宏定义的例子。

### 例程 5-2 字符串是表达式的宏定义。

```
#include <stdio.h>
#define A x*x*x
#define B y*y*y
int main(void)
{
    int x,y;
    printf("Please input x&y\n");
    scanf("%d %d",&x,&y);
    printf("The result of 2x^3+5y^3 is %d",3*A+5*B);
    getchar();
    return 0;
}
```

本例程中将字符串`x*x*x`表示为A,将字符串`y*y*y`表示为B,在进行预处理时,先做宏展开,也就是将源程序中的A替换为`x*x*x`,将源程序中的B替换为`y*y*y`,然后再对源程序进行编译处理。这样,表达式`3*A+5*B`在预处理时经宏展开后变为`3*x*x*x+5*y*y*y`。最终本例程的运行结果为:

```
Please input x&y
2 3
The result of 2x^3+5y^3 is 159
```

必须注意的是,这里的字符串加括号与不加括号是不一样的。像上例中不加括号,预处理之后,表达式`3*A+5*B`变为`3*x*x*x+5*y*y*y`;如果在定义字符串时加括号,即:

```
#define A (x*x*x)
#define B (y*y*y)
```

则预处理之后,表达式`3*A+5*B`就变为`3*(x*x*x)+5*(y*y*y)`,因为宏代换是将标识符后面的表达式作为一个字符串整体代换的。在本例中,无论加不加括号结果都是一样的。但是如果要有运算优先级的差别,加括号与不加括号会影响预处理的结果,进而影响整个程序运算的结果。例如:

```
#define X (a+b)
```

在程序中:

```
int main(void)
{
    ...
    Y=3*X;
    ...
}
```

这样宏代换后表达式`Y=3*X`变为`Y=3*(a+b)`,如果定义时不加括号,即:

```
#define X a+b
```



宏代换后表达式  $Y=3*X$  变为  $Y=3*a+b$ 。显然这两个表达式的运算结果是不一样的。因此在做宏定义时必须注意，应保证在宏代换之后程序不发生错误。

### 5.3.2 宏定义的嵌套

宏定义也是可以嵌套的，即在进行宏定义时，可以直接引用已定义的宏名。看下面这个例子。

#### 例程 5-3 嵌套的宏定义。

```
#include <stdio.h>
#define PI 3.1416
#define S PI*r*r
int main(void)
{
    float r;
    printf("Please input radius\n");
    scanf("%f",&r);
    printf("%f",S);
    getchar();
    return 0;
}
```

本例程中先做宏定义 `#define PI 3.1416`，然后再嵌套地进行宏定义 `#define S PI*r*r`，因为 `PI` 已在之前定义，这里直接引用宏名。在预处理时，将源程序中的 `PI` 宏代换为 `3.1416`，将源程序中的 `S` 宏代换为 `PI*r*r`，最终代换为 `3.1416*r*r`。因此本例程中，语句 `printf("%f",S)` 预处理后变为 `printf("%f",3.1416*r*r)`，再进行编译等处理。本例程的运行结果为：

```
Please input radius
2
12.566400
```

### 5.3.3 宏定义的其他应用

除此之外，还可以应用宏定义来定义数据类型、输出格式等。例如应用宏定义来定义数据类型：

```
#define INTEGER int
```

这样就可程序中用标识符 `INTEGER` 代替关键字 `int`，方便那些用惯了其他编程语言的程序员。再例如应用宏定义来定义输出格式：

```
#define D "%d\n"
```

这样在源程序中，输出语句：

```
printf("%d\n",x);
```

就可以写成：

```
printf(D,x);
```

这样无疑减少了书写麻烦，特别是在一个有很多同样输出格式的程序中，对输出格式采用宏定义会减少代码量，使得程序更加简洁直观。



还要注意一点，`#define` 命令（预处理命令和宏定义命令）必须写在函数之外，宏定义的作用域为宏定义命令起到源程序结束为止。如果要提前终止其作用域，可使用 `#undef` 命令。`#undef` 命令的一般形式为：

`#undef 标识符`

表明到此为止该标识符不再代表宏定义中的字符串，即该标识符在后续程序中无效。

例如：

```
#define PI 3.1416
int main(void)
{
    .....
}
#undef PI
func()
{
    .....
}
```

表示 `PI` 只在 `main` 函数中有效，而在函数 `func` 中无效。

以上介绍了不带参数的宏定义，在实际编程中可以灵活使用宏定义来定义常量、表达式、类型等。这不但使程序更加简洁、清晰，而且提高了代码的可读性和程序的通用性。特别是在一些多参数的程序设计中（例如接口程序设计），会使用到很多数据，这就可以利用宏定义的方法将一些数据定义成有意义的字符常量，提高代码的可读性。另外，将一些常用的固定的参数定义成字符常量存放在库文件中，也便于随时引用，减少了查阅手册的麻烦。

## 5.4 带参数的宏定义

上面介绍了不带参数的宏定义，除此还有一类带参数的宏定义。带参数的宏定义作用很像函数，但本质又与函数不同，且执行效率较函数调用高。本节将详细介绍带参数的宏定义。

### 5.4.1 带参数的宏定义的一般形式

C 语言中允许带参数的宏调用。它的一般形式为：

`#define 宏名 (参数表) 字符串`

与函数类似，带参数的宏定义中的参数称为形式参数，而在程序中宏调用中的参数称为实际参数。与不带参数的宏调用不同，在预处理操作中，不但要进行宏展开，而且要用宏调用中的实参去代换宏定义中的形参。下面通过一个实例来理解带参数的宏定义。

#### 例程 5-4 带参数的宏定义的使用。

```
#include <stdio.h>
#define PI 3.1416
#define C(r) 2*PI*r
int main(void)
```



```

{
    double r=10;
    double circle;
    circle=C(r);
    printf("%lf",circle);
    getchar();
    return 0;
}

```

本例程中应用带参数的宏定义`#define C(r) 2*PI*r`。在预处理时不但要用`2*PI*r`代替程序中的`C(r)`，还要用宏调用中的实参`r`的值`10`去代替宏定义中的形参`r`。综合上述两步，源程序中的`C(r)`就被`2*3.1416*10`所代替。图5-1形象地表明了宏展开和参数置换的过程。

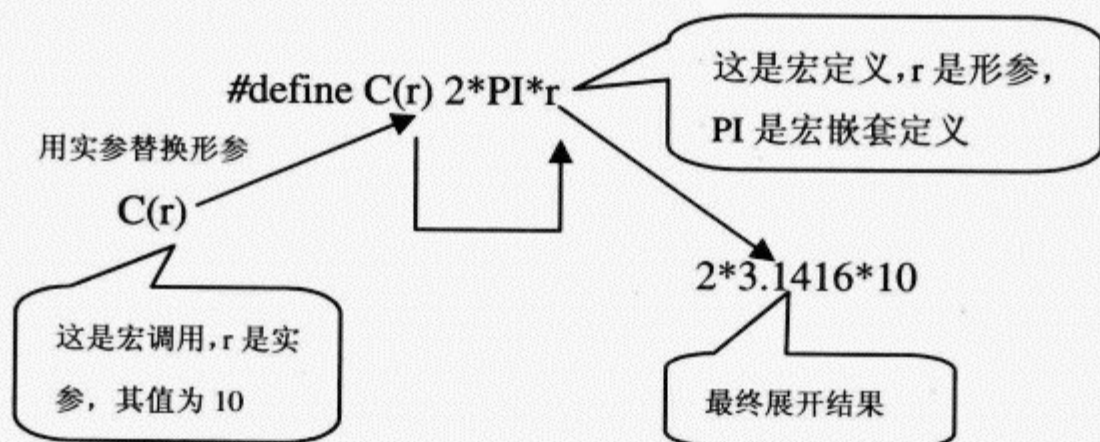


图 5-1 宏展开和参数置换的过程

本例程的运行结果为：

```
62.832000
```

#### 5.4.2 带参数的宏定义与函数

有时带参数的宏定义可以代替函数使用，它与在程序中调用函数的效果是相同的，但两者本质不同。下面通过两段程序来理解带参数的宏定义与函数调用之间的差别。

##### 例程 5-5 带参数的宏定义实现函数的功能。

```

#include <stdio.h>
#define MAX(a,b) (a>b)?a:b
int main(void)
{
    int a,b,max;
    printf("Please input 2 digit\n");
    scanf("%d %d",&a,&b);
    max=MAX(a,b);
    printf("The max is:%d",max);
    getchar();
    return 0;
}

```

##### 例程 5-6 应用函数实现。

```

#include <stdio.h>
int main(void)
{
    int a,b,max;
    printf("Please input 2 digit\n");
}

```



```

scanf("%d %d",&a,&b);
max=MAX(a,b);
printf("The max is:%d",max);
getchar();
return 0;
}
int MAX(int a,int b)
{
    return (a>b)?a:b;
}

```

上面两段程序都是实现在两个数中选择较大的数，但所用的方法不同。例程 5-5 应用带参数的宏定义 `#define MAX(a,b) (a>b)?a:b`，在预处理时，首先将宏调用 `MAX(a,b)` 展开为 `(a>b)?a:b`，在展开的过程中进行实参与形参的置换，即将实参中的 `a`、`b` 的值代到形参中去。最终宏展开和参数置换的结果为：

```
max=(a>b)?a:b;
```

其中 `a`、`b` 的值有待程序的输入。

然而在函数调用中，并不存在宏的展开和参数置换等问题，而且它也不是预处理。带参数的宏定义与函数的区别在于：

(1) 函数的调用为形参分配内存空间，而宏的展开和参数置换等都在程序编译前进行，在展开时只是替换，并无内存空间的分配。

(2) 函数的形参与实参之间是值传递，而宏定义中参数的置换是在预处理的替换过程中，并不存在所谓参数的传递。

(3) 函数调用有返回值，宏定义没有返回值。

(4) 函数中，形参和实参的类型要一致。但是宏并不存在类型问题，宏名无类型，参数无类型，展开时只是代入指定的字符串，宏定义时，字符串可以是任何类型的数据。

(5) 宏替换不占用程序的运行时间，只占用编译时间；而函数的调用要占用程序运行的时间。

当然在这里两段程序的运行结果是相同的：

```

Please input 2 digit
2 3
The max is:3

```

### 5.4.3 使用带参数的宏定义的注意事项

必须注意的是，并非在任何时候宏定义与函数调用都是等效的，有时同一表达式用函数处理与用宏处理两者的结果有可能不同。下面通过两段程序分析一下这种情况。

#### 例程 5-7 应用带参数的宏定义。

```

#include <stdio.h>
#define S(y) ((y)*(y))
int main(void)
{
    int i=1;
    while(i<=5)
        printf("%d\n",S(i++));
}

```



```
    getchar();  
    return 0;  
}
```

### 例程 5-8 应用函数调用。

```
#include <stdio.h>  
int main(void)  
{  
    int i=1;  
    while(i<=5)  
        printf("%d\n",S(i++));  
    getchar();  
    return 0;  
}  
S(int y)  
{  
    return((y)*(y));  
}
```

第一段程序采用带参数的宏定义的方法将  $S(y)$  定义为  $((y)*(y))$ ，计算  $y$  的平方，第二段程序则采用函数调用的方法，返回  $y$  值的平方，但是两段程序的运行结果并不相同。

第一段程序的运行结果为：

```
2  
12  
30
```

第二段程序的运行结果为：

```
1  
4  
9  
16  
25
```

下面具体分析这两段程序的运行过程。

例程 5-7 中采用带参数的宏定义的方法，因此在预处理时要进行宏展开和参数置换。这样

$S(i++)$  就展开成  $((i++)*(i++))$ ，再进行程序的编译、链接，最后生成可执行程序。在程序执行过程中， $i$  的初值为 1，参与运算后自增加 1。循环情况如下：

第一次循环中，第一个括号中  $i$  值为 1，第二个括号中  $i$  值为 2，得到结果 2，然后  $i$  再次自增加 1；

第二次循环中，第一个括号中  $i$  值为 3（上次循环后  $i$  已变为 3），第二个括号中  $i$  值为 4，得到结果 12，然后  $i$  再次自增加 1；

第三次循环中，第一个括号中  $i$  值为 5（上次循环后  $i$  已变为 5），第二个括号中  $i$  值为 6，得到结果 30，然后  $i$  再次自增加 1；

第四次循环判断， $i$  值为 7（上次循环后  $i$  已变为 7），不符合循环条件，循环结束。所以，该程序实际只循环 3 次。

例程 5-8 中采用函数调用的方法，在程序执行过程中，实参先将  $i$  值传递给形参，之后  $i$  自增加 1。循环情况如下：

第一次循环中，实参将 1 传递给形参，之后  $i$  自增加 1。函数内部两个  $y$  值均为 1，返



回结果为 1;

第二次循环中, 实参为自增后  $i$  的值 2, 并将 2 传递给形参, 之后  $i$  再次自增加 1, 返回结果为 4;

.....

如此循环下去, 共循环 5 次。

对比两例不难看出, 并非在任何情况下带参数的宏定义都可以等效于函数调用, 因为其实现的原理存在本质差异, 所以在使用时要加以区分, 审慎使用。

还要注意一点, 在使用带参数的宏定义时, 宏名和形参表之间不能有空格出现。否则在预处理时系统会产生错误理解, 从而导致宏展开的错误。

例如将

```
#define MAX(a,b) (a>b)?a:b
```

写为

```
#define MAX(a,b) (a>b)?a:b
```

将被认为是无参数宏定义, 即宏名为 **MAX**, 代表字符串  $(a,b) (a>b)?a:b$ 。宏展开时, 宏调用语句

```
max=MAX(x,y);
```

将变为

```
max=(a,b) (a>b)?a:b(x,y);
```

这显然是错误的。

以上介绍了带参数的宏定义的用法及使用时的注意事项。在编程时, 灵活地使用带参数的宏定义去替代函数调用, 可以减少在程序执行时因调用函数而导致的系统开销。不过也不提倡过度使用, 因为这样有可能破坏程序的结构性, 而且使用不当可能会产生错误。

## 5.5 文件包含

相信读者对“文件包含”这个字眼已不陌生, 在前面的章节中我们已多次用到。文件包含处理是指在一个源程序文件中, 通过文件包含命令将另外一个源文件的内容全部包含到此文件中。这样在源文件编译时, 连同被包含进来的文件一同编译, 生成一个目标文件。

### 5.5.1 文件包含命令的一般形式

C 语言中提供了 `#include` 命令来实现文件包含操作。具体地有两种形式:

```
#include <文件名>
```

或者

```
#include "文件名"
```

上面两种形式都是合法的, 它们的区别在于, 第一种形式用“<>”预处理时, 系统直



接到存放 C 库函数头文件所在的目录下去寻找所要包含的文件；第二种形式用“`""`”预处理时，系统先在用户的当前目录下寻找要包含的文件，如果找不到要包含的文件，再到存放 C 库函数头文件所在的目录下去寻找。一般地，若要包含的文件是用户自己编写的（一般情况下用户编写的文件都应存放在当前的目录下），最好使用第二种形式的文件包含；若要包含的文件是库函数，则用第一种形式的文件包含，这样系统会直接找到所要包含的文件，节省了查找的时间。

图 5-2 形象地说明了文件包含的关系。

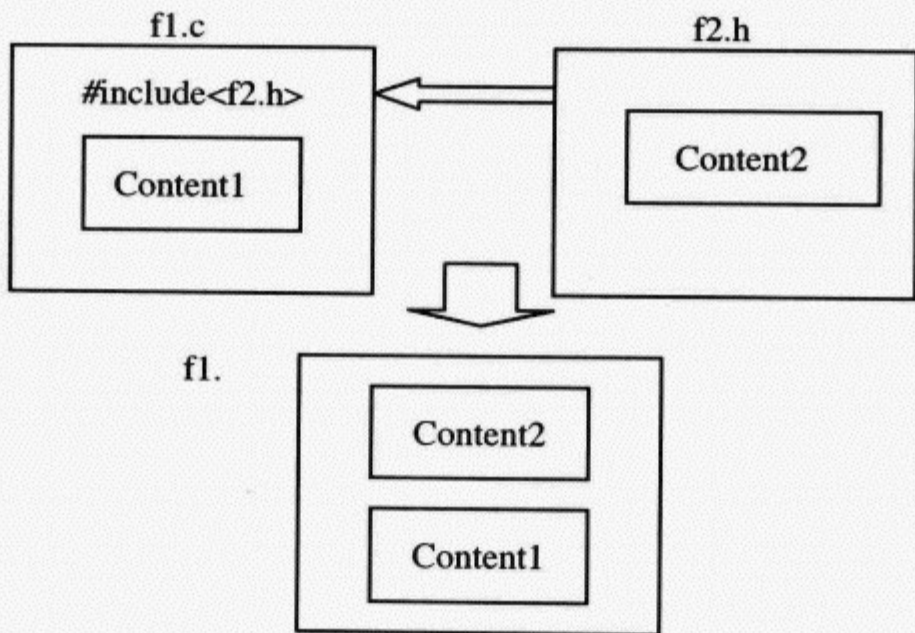


图 5-2 文件包含的示意图

如图 5-2 所示，文件 `f1.c` 中应用 `#include` 命令将文件 `f2.h` 包含在内。其中，`Content1` 为文件 `f1.c` 中的具体内容，`Content2` 为文件 `f2.h` 中的具体内容。在编译预处理时，系统会自动对 `#include` 命令进行处理。具体的做法就是将 `f2.h` 中的 `Content2` 复制到 `#include<f2.h>` 处，这样 `f2.h` 就被包含在 `f1.c` 中了，得到下面的新的 `f1.c`。然后再进行源文件的编译。在编译中，系统会将新的 `f1.c` 视为一个源文件单位进行编译。

文件的包含不仅可以包含 `.h` 文件，还可以包含 `.c` 文件，这种包含同运行多文件程序不一样。运行多文件程序是在源程序编译时把多个文件编译、链接成一个统一的可执行文件，然后运行。而这里所讲的文件包含是发生在预处理阶段，`#include` 是一种预处理命令。在源程序编译之前，系统将所要包含的文件复制到 `#include` 命令处，从而形成一个新的源文件，然后再进行编译。

### 5.5.2 文件包含的特点

前面讲到，文件包含与运行多文件程序是不一样的。文件的包含操作发生在源程序编译之前，因此，文件的包含有如下特点。

#### （1）文件包含的顺序有讲究

如果文件 `f1` 要包含文件 `f2`，而文件 `f2` 要包含文件 `f3`，可以只在文件 `f1` 中插入包含命令，文件 `f2` 中可不插入包含命令。但要注意文件的包含顺序，先包含文件 `f3`，后包含文件 `f2`。



例如文件 f1 中的内容:

```
#include<f3.c>
#include<f2.c>
```

这样, 文件 f1 可使用 f2、f3 的内容, 文件 f2 可使用文件 f3 的内容。即后包含的文件可使用先包含的文件中的内容。

### (2) 允许包含的嵌套

在一个被包含的文件中也可以包含其他文件。

例如文件 f1 中的内容:

```
#include<f2.c> /*f1.c 中包含 f2.c*/
int main(){
... ..
}
```

文件 f2 中的内容:

```
#include<f3.c> /*f2 中包含 f3.c*/
void func(){
... ..
}
```

### (3) 省略外部变量声明 extern

因为多个文件在通过 `#include` 命令预处理后形成一个新的源文件, 而并非在程序编译时才链接在一起。因此, 一个文件中定义的外部变量 (全局变量) 是可以被多个文件共同使用的, 在其他文件中不需要对引用外部变量作 `extern` 声明。看下面这个例子。

#### 例程 5-9 应用文件的包含省略外部变量声明。

文件 5-9-1.c 中的内容:

```
int L; /*周长*/
int S; /*面积*/
#include"5-9-2.c"
int main()
{
    int x,y; /*矩形长宽*/
    scanf("%d %d",&x,&y);
    func(x,y);
    printf("%d,%d",L,S);
}
```

文件 5-9-2.c 中的内容:

```
void func(int x,int y)
{
    L=2*(x+y);
    S=x*y;
}
```

由于函数只能返回一个值, 而这里要同时得到矩形的周长和面积, 因此用全局变量带回。在文件 5-9-1.c 中包含了文件 5-9-2.c, 因此预处理后, 文件 5-9-2.c 的内容插入到命令 `#include 5-9-2.c` 处, 相当于该文件合成为:

```
int L; /*周长*/
int S; /*面积*/
```



```
void func(int x,int y)
{
    L=2*(x+y);
    S=x*y;
}
int main()
{
    int x,y; /*矩形长宽*/
    scanf("%d %d",&x,&y);
    func(x,y);
    printf("%d,%d",L,S);
}
```

然后再进行编译。因此在文件 5-9-2.c 中,虽然没有直接定义全局变量 L 和 S,但可直接引用,不需要再用关键字 `extern` 作外部变量声明。

值得注意的是, `#include 5-9-2.c` 必须置于全局变量 L 和 S 的声明之后,因为在文件 5-9-2.c 中要直接引用这两个全局变量,否则系统编译时按变量未定义错误处理。

本例程的运行结果是:

```
2 3
10,6
```

## 5.6 条件编译

程序员书写的程序,系统要经过预处理、编译、链接等几步处理,最终生成可执行文件。一般地,经过预处理后,源程序中每一行都要进行编译处理,除此之外 C 语言还提供条件编译的功能。简单地讲就是不必对源程序中每一行都进行编译,而是根据程序员指定的条件,对源程序中的一部分代码行进行编译处理。因为有时源程序并不是每一条语句都执行,如果不执行的语句过多而都对它们进行编译的话,会影响整个程序的运行时间。

### 5.6.1 条件编译命令的一般形式

条件编译命令的一般形式如下。

第一种形式:

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

第二种形式:

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

第三种形式:

```
#if 表达式
    程序段 1
```



```
#else
    程序段 2
#endif
```

下面分别介绍。

(1) 第一种形式:

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

作用是如果标识符被宏定义,则在编译时只编译程序段 1,否则只编译程序段 2。当然, #else 部分也可以没有,例如:

```
#if 表达式
    程序段 1
#endif
```

其中,程序段既可以是语句,也可以是命令行。

程序段是命令行的情况:

```
#ifdef INTEL_X86
#define INTEGER_SIZE 16 /*程序段是命令行*/
#else
#define INTEGER_SIZE 32 /*程序段是命令行*/
#endif
```

程序段是语句的情况:

```
#ifdef DEBUG
printf("%d",debuger); /*程序段是语句*/
#endif
```

以上两种形式都是合法的。

(2) 第二种形式:

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

理解了第一种形式的条件编译就不难理解第二种形式了,它只是在命令的第一行将 #ifdef 改为 #ifndef。其作用是:如果标识符未被宏定义,则在编译时只编译程序段 1,否则只编译程序段 2。当然, #else 部分也可以没有,例如:

```
#if 表达式
    程序段 1
#endif
```

其中,程序段既可以是语句,也可以是命令行。例如:

```
#ifndef INTEL_X86
#define INTEGER_SIZE 32 /*程序段是命令行*/
#else
#define INTEGER_SIZE 16 /*程序段是命令行*/
#endif
```



程序段是语句的情况：

```
#ifndef RUN
printf("%d",debuger); /*程序段是语句*/
#endif
```

以上两种形式都是合法的。且作用与上面第一种形式一样。

(3) 第三种形式：

```
#if 表达式
    程序段 1
#else
    程序段 2
#endif
```

它的作用是如果表达式真值为 1（非 0），则在编译时只编译程序段 1，否则只编译程序段 2。因此可以使程序在不同条件下完成不同的功能。

## 5.6.2 条件编译的应用

条件编译对提高 C 程序的通用性很有好处。例如条件编译程序段是命令行的情况：

```
#ifdef INTEL_X86
#define INTEGER_SIZE 16 /*程序段是命令行*/
#else
#define INTEGER_SIZE 32 /*程序段是命令行*/
#endif
```

在编写程序时，如果目标机是 IntelX86 系列，程序前面有类似于如下的定义：

```
#define INTEL_X86
```

来说明计算机的系统时，则它的整型数表示为 16 位（bits）。程序预编译时，常量 INTEGER\_SIZE 定义为 16，因为 IntelX86 系列机的整数用 16 位（bits）表示；否则就表示该处理器不是 IntelX86 系列，则它的整型数表示为 32 位（bits），预编译时，常量 INTEGER\_SIZE 定义为 32。根据不同的机型选择不同的定义，程序就可以适用于不同的平台而不必做任何修改。

当然这种用法多为开发一些复杂的、跨平台程序使用。要想深入理解还需要具体的编程实践，这里只是做一个简单的介绍。编程中最为常用的还是程序段是语句的情况。下面通过一个实例来理解。

### 例程 5-10 条件编译在调试程序时的应用。

```
#define DEBUG /*定义当前为程序调试阶段*/
int main()
{
    int sort[5],i,j,k,t;
    printf("Please input 5 integer:\n");
    for(i=0;i<5;i++)
        scanf("%d",&sort[i]);
    for(i=0;i<4;i++) {
        for(j=0;j<4-i;j++)
            if(sort[j]>sort[j+1])
            {t=sort[j];
              sort[j]=sort[j+1];
              sort[j+1]=t;
            }
    }
}
```



```
    }  
    #ifdef DEBUG /*调试程序段，显示每一趟排序结果*/  
    printf("step: %d\n", i+1);  
    for(k=0; k<5; k++)  
        printf("%d ", sort[k]);  
    printf("\n");  
    #endif  
}  
    printf("The sort result is\n");  
    for(k=0; k<5; k++)  
        printf("%d ", sort[k]);  
}
```

这是一个经典的未被改进的“冒泡排序”程序，是将终端输入的5个数按照从小到大的顺序重新排列起来。这里面用到了数组的相关知识，这里不做具体介绍，但先要对“冒泡排序”的过程有一个简单了解，然后从中体会到条件编译在调试程序时的作用。

“冒泡排序”法是一种经典的排序方法，其基本思想是：一趟排序将最大的（或最小的）数（或待排序对象）逐一置换到本趟排序队列尾部； $n$ 个数（或待排序对象）至少需要 $n-1$ 趟排序。例如一个待排序列9、7、3、5、0的每一趟排序过程如下（从小到大排序）。

- ◇ 第一趟：7、3、5、0、9。
- ◇ 第二趟：3、5、0、7、9。
- ◇ 第三趟：3、0、5、7、9。
- ◇ 第四趟：0、3、5、7、9。

第一趟排序将最大数9置换到本趟排序队列尾部，形成一个序列；第二趟排序将余下的数中最大数7置换到本趟排序队列尾部，形成一个序列；第三趟排序将余下的数中最大数5置换到本趟排序队列尾部，形成一个序列；第四趟排序将余下的数中最大数3置换到本趟排序队列尾部，形成最终排序结果。

这只是一个未被改进的冒泡排序过程示意，在这里并不对排序的算法实现作过多介绍，有关冒泡排序的详细阐述请参看算法分析或数据结构等书目。

当然，排序的目的是为了得到最终的排序序列，即第四趟的结果。然而在调试程序时，往往希望得到每一趟排序的结果。这样一旦程序有错误，就可以从每一趟排序的结果中发现规律，从而方便找出错误的原因。倘若只得到了一个最终结果，就很难确定出错的位置。另外，由于排序时输入的待排序列是随机的，因此程序本身可能有错，但仅在一次排序中恰好与正确结果重合，也就是“程序蒙对了”。如果在调试时得到每一趟排序的结果，且每一趟排序结果都是正确的，就可以更大程度保证源程序的正确性。

本例程中应用条件编译设置了一段程序调试代码，即在调试阶段显示每一趟排序的结果。当程序不需要调试时，只需删掉源程序中`#define DEBUG`的命令，程序运行时就不显示每一趟排序的结果了。

调试状态下本程序的运行结果为：

```
Please input 5 integer:  
9 7 3 5 0  
step1:  
7 3 5 0 9  
step2:  
3 5 0 7 9
```

```

step3:
3 0 5 7 9
step4:
0 3 5 7 9
The sort result is
0 3 5 7 9

```

如果删掉源程序中#define DEBUG 的命令, 本程序的运行结果为:

```

Please input 5 integer:
9 7 3 5 0
The sort result is
0 3 5 7 9

```

通过本例程可以看出条件编译在程序调试中是很有用的。  
下面通过一个实例来理解第三种形式的条件编译的应用。

### 例程 5-11 第三种形式的条件编译。

```

#define R 1
#define PI 3.14
int main(){
    float c,r,s;
    printf("input a number:\n");
    scanf("%f",&r);
    #if R
        s=PI*r*r;
        printf("area of round is: %f\n",s);
    #else
        s=r*r;
    printf("area of square is: %f\n",s);
    #endif
    getch();
    return 1;
}

```

本例程中, 应用条件编译判断表达式 R, 若 R 为真 (非 0), 则编译时编译语句为:

```

s=PI*r*r;
printf("area of round is: %f\n",s);

```

若 R 为非真 (0), 则编译时编译语句为:

```

s=r*r;
printf("area of square is: %f\n",s);

```

在本例中, 宏定义 R 为 1, 因此编译时编译第一句。本例程的运行结果为:

```

input a number:
3
area of round is: 28.260000

```

这里要注意, 它不是条件语句, 而是条件编译。条件语句在编译时表达式真值并不确定, 要在程序执行时才能确定运行哪条分支, 因此, 条件语句每条分支都要编译; 而条件编译则是在编译时就确定表达式真值, 因此根据表达式的真值只编译一条分支。

采用条件编译, 可以减少被编译的语句, 从而减少目标程序的长度, 减少程序的运行时间。如果条件选择的程序段很长, 采用条件编译的方法是十分必要的。



## 5.7 本章小结与要点回顾

本章介绍了预处理命令 (preprocessor directives)。它是 ANSI C 标准规定的一种 C 程序控制命令。预处理工作主要由预处理程序负责完成。本章主要讨论了以下 3 种：

- ✧ 宏定义。
- ✧ 文件包含。
- ✧ 条件编译。

### 1. 宏定义

宏定义分为两种：不带参数和带参数。

#### (1) 不带参数的宏定义

不带参数的宏定义的一般形式为：

```
#define 标识符 字符串
```

其中 `#define` 为宏定义命令，“标识符”为所定义的宏名。“字符串”可以是常数、表达式、格式串等。源程序中出现的宏名都要被替换为宏中定义的字符串，这个过程称为“宏代换”或“宏展开”。

如果要提前终止其作用域，可使用 `#undef` 命令。`#undef` 命令的一般形式为：

```
#undef 标识符
```

表明到此为止该标识符不再代表宏定义中的字符串，即该标识符在后续程序中无效。

#### (2) 带参数的宏定义

带参数的宏定义的一般形式为：

```
#define 宏名(参数表) 字符串
```

宏定义中的参数称为形式参数，程序中宏调用中的参数称为实际参数。在预处理操作中，不但要进行宏展开，而且要用宏调用中的实参去代换宏定义中的形参。

在使用带参数的宏定义时要注意宏调用与函数调用的区别。它们有着本质的不同，而且并非在任何时候宏定义与函数调用都是等效的，有时同一表达式用函数处理与用宏处理两者的结果有可能不同。

### 2. 文件包含

C 语言中提供了 `#include` 命令来实现文件包含操作。具体地有两种形式：

```
#include <文件名>
```

或者

```
#include "文件名"
```

其中用“< >”时，预处理时系统直接到存放 C 库函数头文件所在的目录下去寻找所要包含的文件，而用“””时，预处理时系统先在用户的当前目录下寻找要包含的文件，

如果找不到要包含的文件，再到存放C库函数头文件所在的目录下去寻找。

使用文件包含有如下特点。

(1) 文件包含顺序有讲究：后包含的文件可使用先包含的文件中的内容。

(2) 允许包含的嵌套：在一个被包含的文件中也可以包含其他文件。

(3) 省略外部变量声明 **extern**：应用文件包含可使一个文件中定义的外部变量（全局变量）被多个文件共同使用的，在其他文件中不需要对引用外部变量作 **extern** 声明。

### 3. 条件编译

条件编译命令有三种形式。

第一种形式：

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

作用是如果标识符被宏定义，则在编译时只编译程序段 1，否则只编译程序段 2。当然，**#else** 部分也可以没有。

第二种形式：

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

它只是在命令的第一行将 **#ifdef** 改为 **#ifndef**。其作用是：如果标识符未被宏定义，则在编译时只编译程序段 1，否则只编译程序段 2。

第三种形式：

```
#if 表达式
    程序段 1
#else
    程序段 2
#endif
```

作用是如果表达式真值为 1（非 0），则在编译时只编译程序段 1，否则只编译程序段 2。

灵活地使用条件编译，可以减少被编译的语句、目标程序的长度和程序的运行时间，有利于程序的可移植性。



前面讲过，C 语言中的数据类型包括基本类型、构造类型、指针类型和空类型。其中在第 2 章已经详细介绍了数据类型的 4 种基本类型。本章将讨论构造类型中的数组类型以及指针类型。

## 6.1 数组的概念

数组是有序数据的集合。在程序设计中，为了数据的操作方便，往往把同一类型的数据按一定形式有序地组织起来，这些有序数据的集合就称为数组。在 C 语言中，一个数组有一个统一的数组名，数组中的每一个元素有一个确定的下标来标识。

数组属于构造数据类型，要求数组中的每一个数据元素类型相同。数据元素既可以是基本类型（例如：整型、浮点型……），也可以是其他构造类型（例如结构体），还可以是指针类型。因此，数组的类型是很丰富的。

数组从维数上划分可分为一维数组和多维数组。一般情况下，常使用的数组是一维数组和二维数组。

数组作为一种最为简单的数据结构，用途十分广泛。可以用一维数组来存储一个线性序列，以便对该序列进行各种操作；可以用二维数组来存储一个矩阵，从而实现更为复杂的数据处理。

## 6.2 一维数组

一维数组是最简单的一种数组形式，它是一种线性的数据结构。在物理空间分配上，系统在内存中为一维数组分配一段连续的空间。本节将就一维数组的定义、存储形式、元素的引用、一维数组的初始化等知识加以介绍。

### 6.2.1 一维数组的定义

像使用变量一样，要使用数组必须先进行定义。在 C 语言中，一维数组定义的一般形式为：

类型说明符 数组名 [常量表达式]；

“类型说明符”用来说明数组中元素的数据类型。由于数组中数据的类型要一致，所以每个数组的类型说明符只有一个，也称为该数组的类型。类型说明符可以是任一种基本数据类型或构造数据类型。“数组名”是用户定义的数组标识符，唯一标识该数组。方括号“[]”中的常量表达式表示数组中数据元素的个数，也称为数组的长度。

例如：

```
int a[10];
```

就表明定义一个整型数组（即数组中的数据元素都为整型），数组名为 `a`，数组可存放 10 个整型数据元素。

像定义一般的变量一样，如果程序中定义了一个数组，当系统对源程序进行编译时就会在内存中开辟“[]”中指定大小的连续空间，用数组名作为该连续空间的名字，用户就可以向分配好的数组空间存入数据了。定义一个数组就相当于开辟一段连续的内存空间作为存放数据的容器，例如 `int a[10]`；定义后内存中的情形如图 6-1 所示。

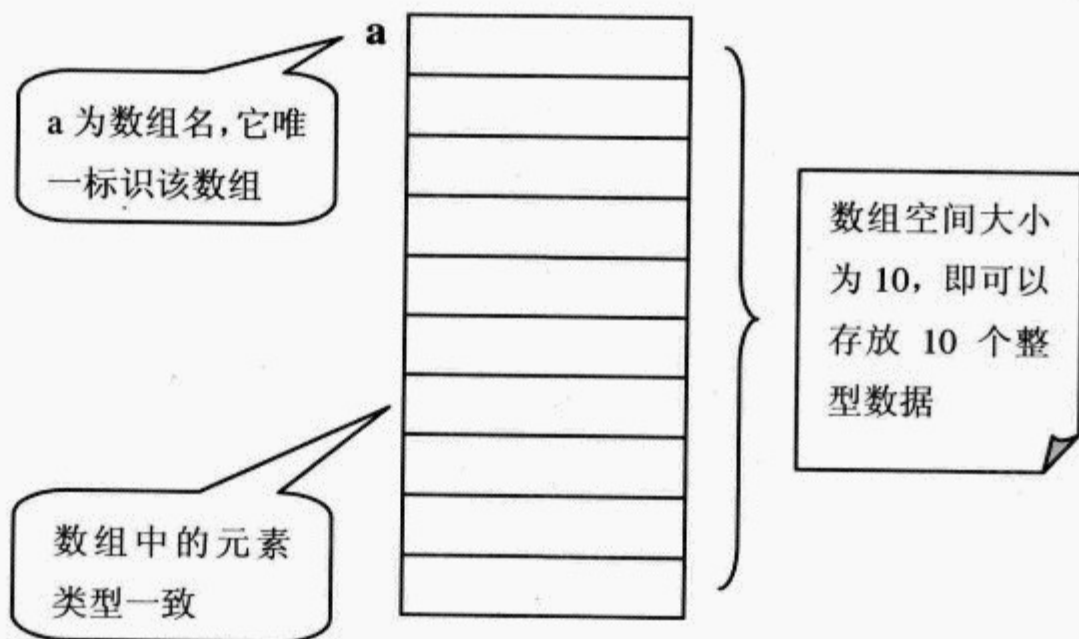


图 6-1 数组在内存中的分配状况

对于数组类型说明要注意以下几点：

(1) 数组的类型实际上是指数组元素的取值类型。同一个数组的所有元素的数据类型都是相同的。

(2) 在编写程序时，数组名的书写规则应符合标识符的书写规定。

例如：

```
int char[5];
```

这种定义数组的方式是不合法的，因为数组名不符合标识符的书写规定，该数组名是一个关键字。

(3) 在编写程序时，数组名不能与其他变量名相同。

例如：

```
int a[10];
```



```
int a;
```

这种定义数组的方式是不合法的，因为数组名与其他变量名相同了。

(4) 方括号中常量表达式表示数组元素的个数，如 `a[5]` 表示数组 `a` 有 5 个元素，但是其下标从 0 开始计算，因此 5 个元素分别为 `a[0]`，`a[1]`，`a[2]`，`a[3]`，`a[4]`。

(5) 不能在方括号中用变量来表示元素的个数，但是可以是符号常数或常量表达式。例如：

```
int a=5;  
char str[a];
```

这种定义数组的方式是不合法的，因为方括号中用变量来表示元素的个数。而

```
#define max 100  
int a[max];
```

这种定义数组的方式是合法的，因为可以用符号常数或常量表达式来表示元素的个数。

## 6.2.2 一维数组的元素

定义了数组就是要使用数组来存储数据、管理数据，这就涉及到数组元素的引用的问题。

数组元素是组成数组的基本单元，对数组元素的引用构成了对数组的基本操作。在 C 语言中，数组元素可表示为：

数组名[下标]

其中下标只能为整型常量或整型表达式。如果是小数，C 编译将自动取整。

例如：

```
a[5]  
a[i+j]  
a[i++]
```

都是合法的数组元素。其中，`a[5]` 表示数组 `a` 中的第 6 个元素；`a[i+j]` 表示数组 `a` 中的第 `i+j+1` 个元素，这里 `i+j` 是确定的值；`a[i++]` 表示数组 `a` 中的第 `i` 个元素，取完该元素后再进行 `i` 的自增 1 操作。

通过下面这个例子来理解对数组元素的引用。

### 例程 6-1 数组元素的引用。

```
#include <stdio.h>  
int main(void)  
{  
    int i,a[10];  
    for(i=0;i<10;i++)  
        scanf("%d",&a[i]);  
    for(i=0;i<10;i++)  
        printf("%d ",a[i]);  
    getchar();  
    return 0;  
}
```

本例程中先定义了一个大小为 10 的整型数组，该数组可存放 10 个整型数据。然

后通过循环语句依次向该数组输入数据。数组下标  $i$  在循环中不断增加，最后一个数组元素为  $a[9]$ 。然后再次通过循环语句将刚才存入数组中的数据读出。在 `scanf` 语句中， $\&a[i]$  为取数组元素  $a[i]$  的地址表达式，这是 `scanf` 函数的参数要求。在 `printf` 语句中， $a[i]$  为直接引用数组元素，目的是显示数组元素  $a[i]$  的值。本例程的运行结果为：

```
3 2 1 4 5 6 7 0 8 9
3 2 1 4 5 6 7 0 8 9
```

注意一点，在 C 语言中只能逐个地引用数组变量，而不能一次引用整个数组。企图一次引用整个数组的操作是不合法的。

例如将下面的语句：

```
for(i=0;i<10;i++)
    printf("%d",a[i]);
```

改写为：

```
printf("%d",a);
```

企图一次输出整个数组的数据元素，这种做法是不合法的。数组中的元素只能逐个引用，对数组元素的引用构成了对数组的基本操作。

另外还要注意区分数组的定义和数组元素的引用。定义数组时不能在方括号中用变量来表示元素的个数，但可以是符号常数或常量表达式。而引用数组元素时，方括号中既可以是常量、常量表达式、字符常数，又可以是变量，不过变量的值一定是确定的。

### 6.2.3 一维数组的初始化

对数组元素赋值时既可以像例程 6-1 那样逐个地引用数组元素变量，然后进行赋值操作，也可以在数组定义时就对数组元素进行赋值，这叫做数组的初始化。

数组初始化是在编译阶段进行的，这样将减少运行时间，提高效率。

对数组元素进行初始化的一般形式为：

```
类型说明符 数组名[常量表达式]={值, 值, ..., 值};
```

也就是在定义数组时给数组元素赋初值。其中“{”中的各数据值即为各元素的初值，它们应是同一类型，各值之间用逗号间隔。

例如：

```
int a[10]={ 0,1,2,3,4,5,6,7,8,9 };
```

就相当于

```
a[0]=0;a[1]=1;...;a[9]=9;
```

以上为对数组的初始化的一般形式，在对数组进行初始化过程中，还有以下几点规定：

(1) 可以只对数组元素部分初始化

例如：

```
int a[10]={1,3,5};
```

在方括号“[]”中定义数组  $a$  有 10 个元素，但花括号“{”中只有 3 个初值。这样，



在系统编译阶段只对该数组的前3个元素赋初值，后7个元素自动补0。

### (2) 数组的一次性赋0

如果想使一个数组中全部元素值为0，除了可以按照常规方法赋初值外，还可以写成

```
int a[10]={0};
```

这样更为简洁方便。

### (3) 可以不指定数组长度

例如：

```
int a[5]={1,2,3,4,5};
```

可写为：

```
int a[]={1,2,3,4,5};
```

这种数组初始化的方法只限于被定义数组长度与提供的初值个数相等的情况，如果它们不相等，则数组的长度不能省略。

## 6.2.4 一维数组举例

应用一维数组可以解决许多实际问题，例如数列的排序问题、字符串操作等都需要数组作为数据的存储结构。如果不用数组作为这些数据的组织结构，实现起来是非常困难和麻烦的。而且一维数组属于顺序线性存储结构，在内存中是一段连续分配的空间，因此它可通过下标直接进行存取，直接找到要找的数组元素，这样的操作效率是很高的。所以，一维数组在实际编程中得到广泛的应用。

下面通过一个一维数组在实际编程中应用来理解一维数组的用法。

在第5章中介绍过一个利用数组进行“冒泡排序”的程序。这里再介绍一个经典的排序算法——“选择排序”，待排序的数列仍然是用数组存储的。

### 例程 6-2 选择排序演示（从小到大排列）。

```
#include <stdio.h>
#define MAX 10
int main(void)
{
    int i,j,t,sort[MAX],min;
    printf("Please input ten integer:\n");
    for(i=0;i<MAX;i++)
        scanf("%d",&sort[i]);
    for(i=0;i<MAX-1;i++)
    {
        min=sort[i];
        for(j=i+1;j<MAX;j++)
            if(sort[j]<min)
            {
                t=min;
                min=sort[j];
                sort[j]=t;
            }
        sort[i]=min;
    }
    printf("The result of sort is:\n");
```

```
for(i=0;i<MAX;i++)
    printf("%d ",sort[i]);
getchar();
return 0;
}
```

这是一个经典的选择排序的算法实现，这个算法还可以改进，使它的执行效率更高，但在这里关心的是一维数组的使用，所以使用一个较为容易理解的算法来举例。

选择排序(selection sort)的核心思想是： $n$ 个元素需要 $n-1$ 趟排序，每一趟排序的操作是选出待排序列中的最小(或最大)元素，再放到本趟排序序列的第一个元素的位置上。下一趟排序从下一个元素开始。这样 $n$ 个元素，恰好需要 $n-1$ 趟排序，最后剩下的第 $n$ 个元素一定是最大(或最小)的。

具体实现选择排序时需要一个中间变量min(或max)，如果是从小到大排序，就需要一个中间变量min。

假设有一组待排序列5、6、3、1、10从小到大排序。

在第一趟的排序中，先将本趟排序序列的第一个位置上的元素sort[0]赋值给min，即假设它是最小的元素，然后将min依次与sort[1]~sort[4]元素进行比较。在比较过程中，如果出现有比min小的元素，就将它与min进行交换，这样保证min中始终是最小的元素。比较结束后将min放到本趟排序序列第一个位置上，这样就完成了一趟选择排序。按照这个步骤，该序列第一趟的排序过程如下。

原序列:	5、6、3、1、10	min=5
循环一次:	5、6、3、1、10	min=5 因为5小于6
循环二次:	3、6、5、1、10	min=3 因为3小于5, sort[2]与min交换
循环三次:	1、6、5、3、10	min=1 因为1小于3, sort[3]与min交换
循环四次:	1、6、5、3、10	min=1 因为1小于10

将min放到本趟排序序列第一个位置上: 1、6、5、3、10。

第一趟排序后，程序选出了原序列中最小的元素1，并将它放在本趟排序序列的第一个位置上。接下来的排序序列就是6、5、3、10了，因为最小的元素1已经选出。按照上面所讲的步骤，后面几趟的排序结果如下。

第二趟排序结果: 1、3、(6、5、10)  
第三趟排序结果: 1、3、5、(6、10)  
第四趟排序结果: 1、3、5、6、10

其中划线的元素为本趟排序中选择出来的元素，括号中的元素为下一趟排序序列中的元素。五个元素的序列只需要四趟排序即可，最后一个剩下的元素一定是最大的。

本例程的运行结果为:

```
Please input ten integer:
23 5 2 7 89 3 1 0 12 4
The result of sort is:
0 1 2 3 4 5 7 12 23 89
```

这里待排序的数列是用数组存储的，这样操作起来十分方便，系统开销也小。当然也



可使用其他数据结构存储数列，但那样操作起来十分麻烦，而且系统的时间开销和空间开销都会很大。所以用数组作为待排序数列的存储结构是十分合适的。

## 6.3 二维数组

在实际问题中有很多数据是二维的或多维的。例如：代数中的矩阵、生产工作中的报表、甚至是图像文件，这些数据都是以二维形式存储的，单靠一维的数组是很难组织这些数据的。在C语言中允许构造多维数组。与一维数组不同，多维数组元素有多个下标，用来标识它在数组中的位置。采用多维数组可以很好地组织多维数据，能够解决更为复杂的问题。在这里只介绍二维数组，有关多维数组的知识可参看数据结构等相关书目。

### 6.3.1 二维数组的定义

二维数组定义的一般形式：

类型说明符 数组名[常量表达式 1][常量表达式 2]

“类型说明符”用来说明数组中元素的数据类型。“数组名”唯一标识该二维数组。第一个方括号“[]”中的常量表达式 1 表示第一维下标的长度，第二个方括号“[]”中常量表达式 2 表示第二维下标的长度。

例如：

```
int a[2][4];
```

说明了一个 2 行 4 列的数组，数组名为 a，每个数组元素的类型均为整型。该数组的数组元素共有  $2 \times 4 = 8$  个，即：

a[0][0],a[0][1],a[0][2],a[0][3]  
a[1][0],a[1][1],a[1][2],a[1][3]

在理解二维数组时，可以就把它看成一个矩阵。不过每一维的下标都是从 0 开始的，即第一行第一列的数组元素下标为 (0, 0)，第 n 行第 n 列的数组元素下标为 (n-1, n-1)。这样在对二维数组进行操作时就不容易出错了。

还可以将二维数组看成一个一维数组，只不过该一维数组中的每个数组元素又是一个一维数组。就像图 6-2 所示那样。

a[0]——a[0][0],a[0][1],a[0][2],a[0][3]  
a[1]——a[1][0],a[1][1],a[1][2],a[1][3]

图 6-2 二维数组示意图

a[0]、a[1]中的元素又是一个一维的 4 个元素的数组。因此，在这里也可把 a[0]、a[1] 看作是一维数组名。

其实在内存中二维数组同一维数组一样，都是线性存储的，它们在内存中占据一块连续的存储空间。二维数组中元素的存放顺序按行存放，即先存放二维数组的第一行，再存

放第二行……例如：定义一个二维数组 `a[2][3]`，定义后内存中的情形如图 6-3 所示。

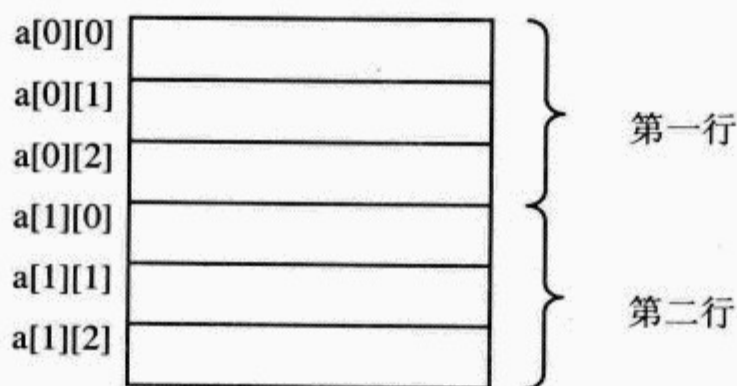


图 6-3 二维数组的内部存储形式

从图 6-3 中可以看出在内存中先存放二维数组的第一行 `a[0][0]~a[0][2]`，再存放二维数组的第二行 `a[1][0]~a[1][2]`。二维数组在逻辑上是二维的，而在实际的物理存储上是一维的。

### 6.3.2 二维数组的元素

与一维数组相同，定义了二维数组后就要用它存储数据、管理数据。二维数组的元素也称为双下标变量，其表示的形式为：

数组名[下标][下标]

其中下标应为整型常量或整型表达式。

例如：

```
a[5][5]
a[i][j]
a[5-2][3*3]
```

都是合法的二维数组元素。

在引用二维数组的元素时，要注意下标表示的范围。例如：定义一个二维数组

```
int a[3][4];
```

该二维数组的下标表示范围为  $(0, 0) \sim (2, 3)$ 。即从 `a[0][0]` 到 `a[2][3]`，因此

```
a[3][4]=2;
```

这样引用数组元素的方法是错误的。

二维数组常用来存储以二维形式表现的数据。下面通过例子来理解二维数组元素的引用和数据的存储。

#### 例程 6-3 用二维数组存储矩阵。

```
#include <stdio.h>
int main(void)
{
    int a[3][4], i, j;
    printf("Please input 3x4 matrix\n");
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            scanf("%d", &a[i][j]);
}
```



```

printf("The matrix is\n");
for(i=0;i<3;i++)
{
    for(j=0;j<4;j++)
        printf("%d ",a[i][j]);
    printf("\n");
}
getchar();
return 0;
}

```

本程序的作用是在内存中开辟一个  $3 \times 4$  大小的空间作为一个二维数组，存放一个 3 行 4 列的整数矩阵。向一个二维数组中输入数据多采用例程中的方法，即应用一个二重循环。外层循环用来控制二维数组的“行”，内层循环用来控制二维数组的“列”。同理输出一个二维数组的数据元素，也可通过一个二重循环逐一输出。本例程的运行结果为：

```

Please input 3x4 matrix
1 2 3 4
5 6 7 8
9 0 1 2
The matrix is
1 2 3 4
5 6 7 8
9 0 1 2

```

### 6.3.3 二维数组的初始化

当然也可以在定义二维数组时对数组进行初始化。有两种方法对二维数组进行初始化。

(1) 按行分段赋值。例如：

```
int a[2][3]={{1,2,3},{4,5,6}};
```

这种初始化方法将二维数组看成一个矩阵，把每一行的元素写在一个花括号“{}”中。注意，每一行元素的花括号之间要有逗号分隔。

(2) 按行连续赋值。例如：

```
int a[2][3]={1,2,3,4,5,6};
```

这种初始化方法将二维数组看成一个连续的空间，其效果与第一种方法相同。但如果要赋值的数据较多，建议使用第一种初始化方法，因为那样更直观。

下面通过实例来理解二维数组的初始化。

#### 例程 6-4 二维数组的初始化。

```

#include <stdio.h>
int main(void)
{
    int a[3][4]={{1,2,3,4},{5,6,7,8},{9,0,1,2}},i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
            printf("%d ",a[i][j]);
        printf("\n");
    }
    getchar();
}

```

```
return 0;
}
```

本例程在定义二维数组 `a[3][4]` 时对它进行初始化。这里使用的是按行分段赋值的方法，然后应用二重循环输出这个二维数组中的元素。本例程的运行结果是：

```
1 2 3 4
5 6 7 8
9 0 1 2
```

在对二维数组进行初始化时有以下几点需要注意。

(1) 同一维数组一样，二维数组在初始化时可以只对部分元素赋初值，未赋初值的元素自动取 0 值。

例如：

```
int a[3][4]={{1},{2},{3}};
```

这样只对该二维数组的第一行第一列赋值 1，第二行第一列赋值 2，第三行第一列赋值 3，其余元素的位置自动取 0。

(2) 如果对全部元素赋初值，则第一维的长度可以不给出。

例如：

```
int a[3][3]={1,2,3,4,5,6,7,8,9};
```

等价于

```
int a[][3]={1,2,3,4,5,6,7,8,9};
```

系统会根据数据的个数自动分配空间。但要注意这种赋值方法只限于对全部元素赋初值的情况。

### 6.3.4 二维数组举例

二维数组的应用十分广泛。本节将使用二维数组来存储代数中常见的矩阵，并通过二维数组对矩阵进行加工处理，实现矩阵的运算。

#### 例程 6-5 应用二维数组实现矩阵的转置运算。

分析：所谓矩阵的转置运算就是将矩阵  $A_{m \times n}$  转化成为矩阵  $A^T_{n \times m}$ ，使得  $A^T_{ij}=A_{ji}$ 。例如矩阵：

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

转置运算后得到矩阵

$$\begin{pmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{pmatrix}$$



下面给出程序代码:

```
#include <stdio.h>
int main(void)
{
    int a[3][4], b[4][3], i, j;
    printf("Please input 3X4 matrix\n");
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            scanf("%d", &a[i][j]); /*输入 3X4 的矩阵*/
    for(i=0; i<4; i++)
        for(j=0; j<3; j++)
            b[i][j]=a[j][i]; /*实现矩阵的转置*/
    printf("The revier matrix is\n");
    for(i=0; i<4; i++)
    {
        for(j=0; j<3; j++)
            printf("%d ", b[i][j]); /*输出转置后的结果*/
        printf("\n");
    }
    getchar();
    return 0;
}
```

本程序用二维数组  $a[3][4]$  存放待输入的矩阵, 用  $b[4][3]$  存放  $a$  转置后的矩阵。首先通过一个二重循环输入一个  $3 \times 4$  阶的矩阵, 然后对该矩阵进行转置操作。转置操作也是通过一个二重循环实现, 只要在循环体内部将  $a[j][i]$  赋值给  $b[i][j]$  即可。最后输出转置后的矩阵  $b[4][3]$ 。本例程的运行结果为:

```
Please input 3X4 matrix
1 2 3 4
5 6 7 8
9 0 1 2
The revier matrix is
1 5 9
2 6 0
3 7 1
4 8 2
```

### 例程 6-6 矩阵的乘法。

在数学中常遇到矩阵相乘的问题, 下面给出两个矩阵:

$$\begin{pmatrix} 1 & 2 & 0 \\ 2 & 3 & 5 \\ 1 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 2 & 2 & 3 \\ 1 & 3 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

编写一个程序, 求解这两个矩阵的乘积。

**分析:** 矩阵的乘积问题是经典的循环嵌套问题。先来讨论矩阵乘法的运算法则。一个  $x \times y$  的矩阵同  $y \times z$  的矩阵相乘, 结果是得到一个  $x \times z$  的矩阵。设矩阵  $A = [a_{ij}]_{x \times y}$ , 矩阵  $B = [b_{ij}]_{y \times z}$ , 则  $A \cdot B = [c_{ij}]_{x \times z}$ 。其中  $c_{ij} = (a_{i1}, a_{i2}, \dots, a_{in}) \cdot (b_{1j}, b_{2j}, \dots, b_{nj})$ 。

从上面的计算法则中不难看出, 要计算两个矩阵的乘积需要三重循环。即:

A 矩阵的第  $i$  行和 B 矩阵的第  $j$  列各元素相乘, 得到  $c_{ij}$ 。

A 矩阵的第  $i$  行分别和 B 矩阵的第  $1 \sim z$  列相乘, 得到  $c_{i1}, c_{i2}, \dots, c_{iz}$ 。

A 矩阵的第  $1 \sim x$  行分别与矩阵的第  $j$  列相乘, 得到  $c_{1j}, c_{2j}, \dots, c_{xj}$ 。

因此, 解决矩阵相乘的问题的一种比较简单的方法就是用三重循环嵌套语句。

下面给出程序代码:

```
#include <stdio.h>
int main(void)
{
    int A[3][3]={{1,2,0},{2,3,5},{1,1,1}};
    int B[3][3]={{2,2,3},{1,3,1},{0,0,1}};
    int C[3][3]={{0,0,0},{0,0,0},{0,0,0}};
    int i,j,k;
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            for(k=0;k<3;k++)
                C[i][j]=C[i][j]+A[i][k]*B[k][j];
    printf("The result is\n");
    for(i=0;i<3;i++) {
        for(j=0;j<3;j++)
            printf("%d ",C[i][j]);
        printf("\n");
    }
    return 0;
}
```

本程序同样用二维数组来存放矩阵, 其中  $A[3][3]$  存放第一个矩阵,  $B[3][3]$  存放第二个矩阵,  $C[3][3]$  存放矩阵 A、B 相乘的结果。在这里使用的是数组的初始化, 然后通过一个三重循环得到矩阵  $C[3][3]$  中的元素, 最后输出结果矩阵  $C[3][3]$ 。本例程的运行结果为:

```
The result is
4 8 5
7 13 14
3 5 5
```

## 6.4 指针的概念

指针是 C 语言中一类重要的数据类型, 也是 C 语言的主要风格之一。C 语言之所以功能强大, 主要就是引入了指针的功能。使用指针可以表示复杂的数据结构, 不需要命名就能够动态分配内存, 还可以直接处理内存地址, 从而编出精练而高效的程序。而且指针型变量也是许多库函数和用户自定义函数的参数, 正确理解和使用指针对学好 C 语言很有帮助。

### 6.4.1 内存的地址

在计算机中, 所有要处理的数据和要运行的程序都要事先放在内存中。而这些数据的存放并非杂乱无章的, 系统将内存划分为一个一个的存储单元, 不同的数据根据其性质和长度不同, 被放到不同的内存单元中。一般把存储器中的一字节 (8bits) 称为一个内存单元, 不同类型的数据所占用的内存单元数也不相同。例如, 一个整型的数据占 2 字节, 因此它就要占据 2 个内存单元; 一个字符型变量占 1 字节, 因此它就要占据 1 个内存单元。为了准确地访问到这些内存单元, 系统为划分好的每个内存单元都编上号, 这样只要根据



一个内存单元的编号便可准确地找到该内存单元了。内存单元的编号就叫做内存的地址。图 6-4 形象地描绘了内存和内存地址。

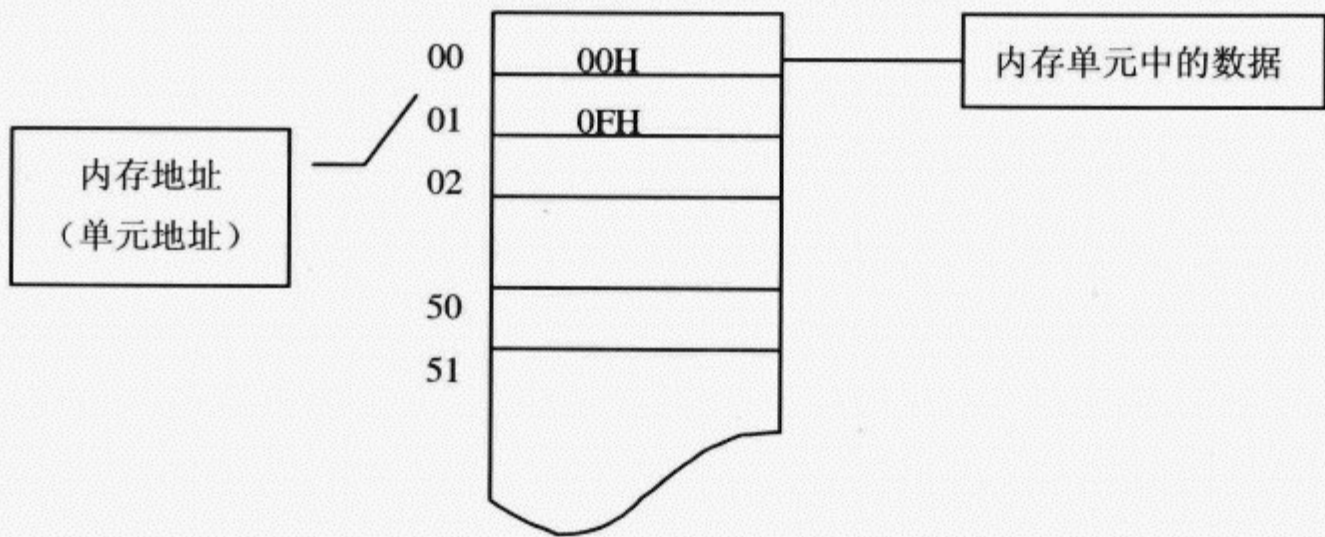


图 6-4  内存和内存地址

从图 6-4 可以看出，内存是一个连续的物理空间，被划分为大小相等的内存单元，每个内存单元都分配一个固定的内存地址，内存地址也是连续的。

内存被划分为一个一个的内存单元，每个内存单元又分配了相应的内存地址之后，就可以根据一个内存单元的地址准确地找到该内存单元了。例如要寻找存储在 00 号单元的整型数据，就可以先找到内存中的 00 号单元，然后从 00 号单元开始顺序读取两字节大小的数据作为该整型数据，也就是 000FH。其实这个整型数据占据了两个内存单元，系统是按字节编址的，因此系统分配 00 和 01 两字节来存放整型数据。

在真实的计算机系统中，为数据分配地址、寻址都是很复杂的工作。在有操作系统的计算机上，程序中给出的地址往往只是内存的逻辑地址，还要通过操作系统的内存管理机制将逻辑地址映射为物理地址。上面的介绍只是将复杂的问题加以简化抽象，这样更加便于理解。其实在这里只要理解什么是内存单元，什么是内存单元的地址，并能区分内存的地址与内存中的数据就可以了。更深一层的有关内存分配、内存管理的知识可以参看操作系统、计算机体系结构等书目。

6.4.2  指针和指针变量

指针就是内存的地址。内存单元的指针和内存单元的内容是两个不同的概念。指针是地址，是内存单元的编号，而内存单元的内容是存放在内存单元中的具体数据。如果为一个变量分配一个内存单元，那么该变量的地址就称为该变量的指针，该变量中的数据就是内存单元中的内容。

在 C 语言中，允许用一个变量来存放指针（内存地址），这种变量称为指针变量。指针变量的值就是某个内存单元的地址，或称为某内存单元的指针。

在这里一定要严格区分变量、变量的指针、变量的内容、指针变量、指针变量的内容这几个概念。它们的关系如图 6-5 所示。

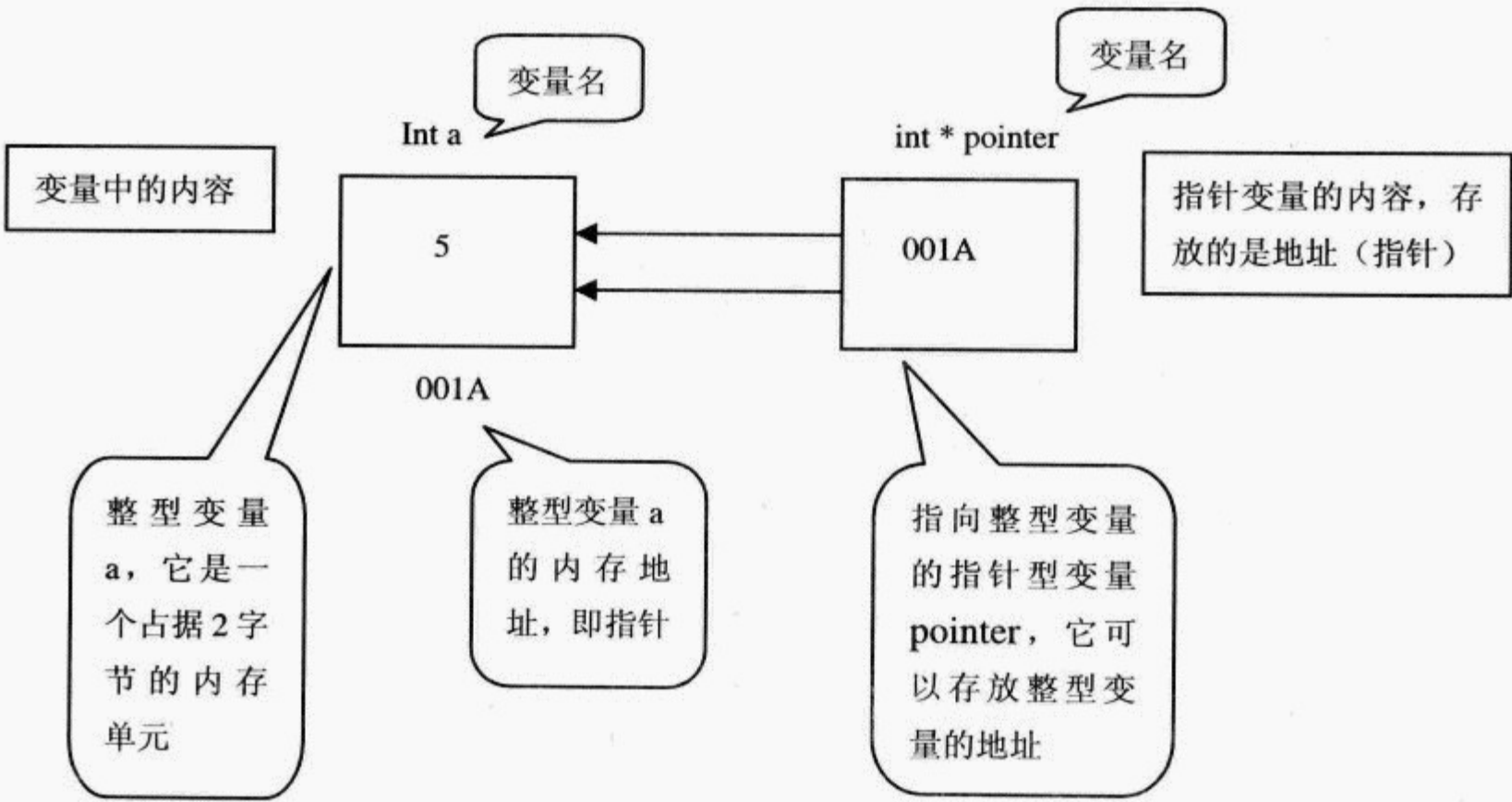


图 6-5 变量与指针变量

从图 6-5 中可以清晰地看出变量、变量的指针、变量的内容、指针变量、指针变量的内容之间的关系。当定义一个变量

```
int a;
```

时，系统在内存为该变量开辟一个空间，该空间的大小为 2 字节，占据两个内存单元。高字节的地址 001A 作为该变量的内存地址，即指针。该整型变量的变量名为 a，是该变量的符号地址，是这个整型变量的标志。通过变量名可以找到变量的内存地址，但变量名本身并不是该变量的内存地址，这一点应当注意。每次向一个变量赋值时，

```
int a;  
a=2;
```

都是通过变量名 a 找到相应的内存地址 001A，然后进行赋值的。这些工作全部由编译系统完成，对程序员是透明的，程序员不必掌握。

pointer 是指向整型变量的指针型变量，它与一般的变量没什么区别，只是它必须存放地址，就如同整型变量必须存放整数一样。pointer 变量中的内容是整型变量 a 的地址，即 a 的指针，这样就称作指针型变量 pointer 指向整型变量 a。

### 6.5 指针型变量的定义

与其他变量一样，指针变量在使用前也需要定义。指针变量也有类型，不同类型的指针变量指向不同类型的变量，也就是说不同类型的指针变量存储不同类型的变量的指针。例如：整型的指针变量只能存放整型类型变量的地址，字符型的指针变量只能存放字符型



变量的地址等。指针变量的类型也称为指针变量的基类型。

对指针变量的定义包括以下三个内容：

- (1) 指针类型说明，即定义该变量为一个指针变量，用“\*”说明。
- (2) 指针变量名。
- (3) 指针变量的类型，即基类型。

其一般形式为：

类型说明符 \*指针变量名；

其中，类型说明符表示指针变量所指向的变量的数据类型，也就是该指针变量的基类型。\*表示这是一个指针变量，有别于其他的变量。变量名即为定义的指针变量名。

例如：

```
int *p;  
char *ch;  
float *ptr;  
FILE *fp;
```

都是合法的定义。第一个变量定义是定义一个指向整型变量的指针变量；第二个变量定义是定义一个指向字符型变量的指针变量；第三个变量定义是定义一个指向浮点型变量的指针变量；第四个变量定义是定义一个指向文件型对象的指针变量。

对于定义好的指针变量，它所能指向的变量的类型也就固定下来了。例如，一个定义为指向整型变量的指针变量，就只能指向整型变量，而不能指向其他类型的变量，如浮点型变量等。

## 6.6 指针型变量的引用

定义了不同类型的指针变量就要用它来指向不同类型的变量，从而对这些变量进行灵活的操作。本节介绍如何应用指针变量操作数据。

### 6.6.1 指针变量引用的方法

对指针变量的赋值与对其他变量的赋值有所不同，指针变量只能存放指针（地址），不能将非指针类的数据赋值给指针变量。例如：

```
int *ptr;  
ptr=100;
```

是非法的，因为 100 是非指针类的数据。

如何才能取得一个变量的地址呢？这就涉及到取地址符“&”，它的作用是取一个变量的地址。例如：

```
&a;
```

就表示变量 a 的地址。因此下列的语句：

```
int *ptr, i;
```

```
ptr=&i;
```

是合法的。它表示定义一个整型变量 *i* 和一个指向整型变量 *ptr*，并把 *i* 的地址 *&i* 赋值给指针变量 *ptr*。

在指针变量的引用中，还有一个重要的运算符“\*”，它叫作指针运算符。作用有两条：一是在定义指针变量时作指针类型说明，表明定义的变量是指针型变量。例如：

```
int *p;
```

表明 *p* 是一个指针型变量。二是用于间接访问指针变量所指向的内存单元。例如：

```
int *p, i=2;  
p=&i;
```

*p* 存放了整型变量 *i* 的地址，*\*p* 为指针 *p* 所指向的内存单元的内容，即 *\*p* 为 2。可以用图 6-6 表明上述关系。

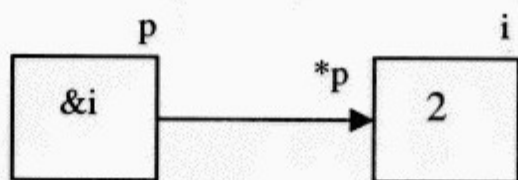


图 6-6 *p*, *\*p*, *i* 的关系

设有指向整型变量的指针变量 *p*，如要把整型变量 *a* 的地址赋予 *p* 可以有以下两种方式。

#### (1) 指针变量初始化的方法

```
int a;  
int *p=&a;
```

在定义指针型变量 *p* 的同时将整型变量 *a* 的地址 *&a* 赋值给 *p*。注意：这里必须有“\*”。

#### (2) 赋值语句的方法

```
int a;  
int *p;  
p=&a;
```

这是以赋值语句的方法将 *a* 的地址 *&a* 赋值给 *p*。注意：这里没有“\*”，因为 *p* 才是指针变量，*\*p* 是 *p* 所指向的内容。

另外，指针变量中的内容也是可以随时改变的，也就是说指针变量可以随时改变指向。例如：

```
int *p1,*p2;  
int i,j;  
p1=&i;  
p2=&j;  
p1=p2;
```

开始指针变量 *p1* 指向整型变量 *i*，指针变量 *p2* 指向整型变量 *j*。但当 *p2* 赋值给 *p1* 后，指针变量 *p2* 也指向了整型变量 *i*。实际上，*p1*, *p2* 都存放着变量 *i* 的地址，*\*p1*, *\*p2* 都是变量 *i* 的内容。



6.6.2 指针应用举例

下面通过一个实例来理解指针变量的引用。

例程 6-7 指针变量的引用。

```
#include <stdio.h>
int main(void)
{
    int *p1,*p2,*t;
    int i,j;
    i=1;
    j=10;
    p1=&i;
    p2=&j;
    /*显示当前的*p1 和*p2 的值*/
    printf("The *p1&*p2 are:%d,%d\n", *p1,*p2);
    t=p1;
    p1=p2;
    p2=t;
    /*显示交换后的*p1 和*p2 的值*/
    printf("The *p1&*p2 are:%d,%d\n", *p1,*p2);
    getchar();
    return 0;
}
```

本例程中先定义了三个指针型变量\*p1、\*p2、\*t，用来指向整型变量。然后定义了两个整型变量i、j，并将i赋值为1，j赋值为10。再将i的地址&i赋值给p1，j的地址&j赋值给p2，即p1、p2分别指向i、j，如图6-7所示。

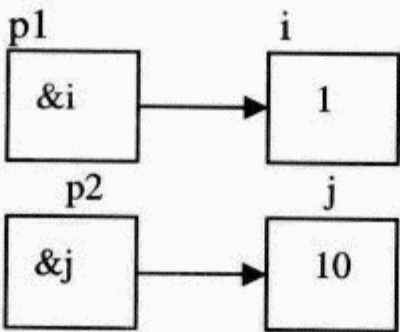


图 6-7 p1, p2 分别指向 i, j

这样\*p1、\*p2的结果分别为1和10。

然后借助中间指针变量t将p1、p2中的内容交换。即让p1指向变量j，让p2指向变量i。p1、p2内容交换后的情形如图6-8所示。

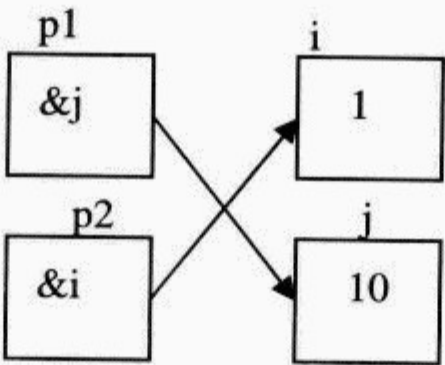


图 6-8 p1, p2 内容交换后的情形

这样\*p1、\*p2的结果分别为10和1。

本例程的运行结果为：

```
The *p1&*p2 are:1,10
The *p1&*p2 are:10,1
```

在使用指针变量时，要特别注意运算符“&”与“\*”的使用顺序，顺序不同，所表达的意思也可能不同。

在C语言中运算符“&”与“\*”的优先级是相同的，因此当两个运算符同时出现时，按照自右而左方向结合。例如：

```
int *p,i=1,*q;
p=&i;
q=&*p;
```

p是一个指针变量，它指向整型变量i，那么q=&\*p是什么意思呢？按照自右而左方向结合的规律来分析一下。

首先进行\*p的运算，它是间接访问指针变量所指向的内存单元运算。也就是说\*p为i的内容，即等于2。再执行&运算，即得到变量i的地址。因此&\*p与&i相同，都为i的地址。可以用图6-9形象地表明这一过程。

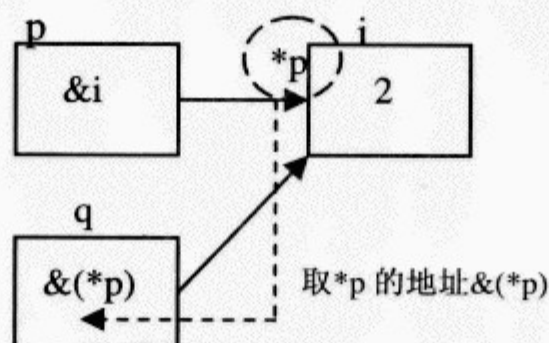


图 6-9 q=&\*p 的过程

如果是将上述的运算符“&”和“\*”反过来，表达的意思又有所不同。例如：

```
int i=2,j;
j=*&i;
```

定义两个整型变量i和j，并将2赋值给i，然后进行j=\*&i运算。同样按照自右而左方向结合，首先计算&i，即取变量i的地址，然后进行\*运算，将地址&i所指向的内容赋值给j。实际上就是把i的内容赋值给j。可以用图6-10形象地表明这一过程。

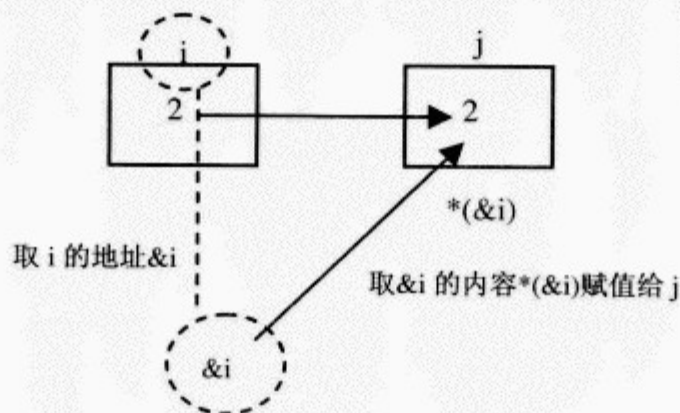


图 6-10 j=\*&i 的过程



在实际编程中，不建议在“\*”和“&”的顺序上作过多的纠缠，有时用一个简单的赋值语句就可以完成的任务大可不必用指针指来指去，这会降低程序的可读性。但深刻理解指针并熟练掌握指针的各种用法很重要，它会使程序员更加灵活地操纵数据。随着编程实践的增多，读者自然能体会到这一点。

## 6.7 指针作为函数参数

指针的引入为C程序设计提供了功能强大的工具，使得程序的编写更加灵活；而函数则是结构化程序设计的灵魂，是C程序设计中所提倡的。要实现二者的完美结合，就需要将指针作为函数的参数。

### 6.7.1 引入

在第4章中已经讲到，函数的参数分为实参和形参两种。在定义函数时，函数的参数表列中的参数称为形参，形参是被调函数中的局部变量。在调用函数时，函数的参数表列中的参数称为实参，实参是主调函数中的局部变量。例如：

```
int main()
{
    ... ..
    func(a,b); /*a,b为实参*/
    ... ..
}
int func(int a,int b)/*a,b为形参*/
{
    ... ..
}
```

调用函数func时的a,b是实参，被调函数中定义的函数参数a,b为形参。实参和形参之间通过值传递传送数据。

但有时正是这种值传递的限制，使得函数很难实现某些功能。看下面这个例子。

#### 例程 6-8 设计一个函数实现两个变量的内容对调。

初学者往往凭直觉写下下面这段程序：

```
#include <stdio.h>
int main(void)
{
    int a=1,b=10;
    printf("a=%d,b=%d\n",a,b);
    /*企图调用 swap 函数实现两数的对调*/
    swap(a,b);
    printf("a=%d,b=%d\n",a,b);
    getchar();
    return 0;
}
int swap(int a,int b)
{
    int t;
    t=a;
    a=b;
```



```
b=t;  
}
```

但是这个 swap 函数并不能像预想的那样实现两数的对调。本程序的运行结果为：

```
a=1,b=10  
a=1,b=10
```

变量 a、b 的内容并没有对调。下面来分析一下造成这种结果的原因。

函数的实参与形参之间的数据传递是值传递，而在被调函数中的形参也只是被调函数的局部变量。因此，在函数调用时，实参将数据传递给形参，也就是说形参只是实参数据的一个拷贝，除此外形参与实参没有任何其他关系。而被调函数中的形参虽然做了对调的操作，但它们只是被调函数中的局部变量，函数执行完毕后就释放掉了，不会对实参产生任何影响，所以才会出现上面的结果。可以用图 6-11 形象地展示这一过程。

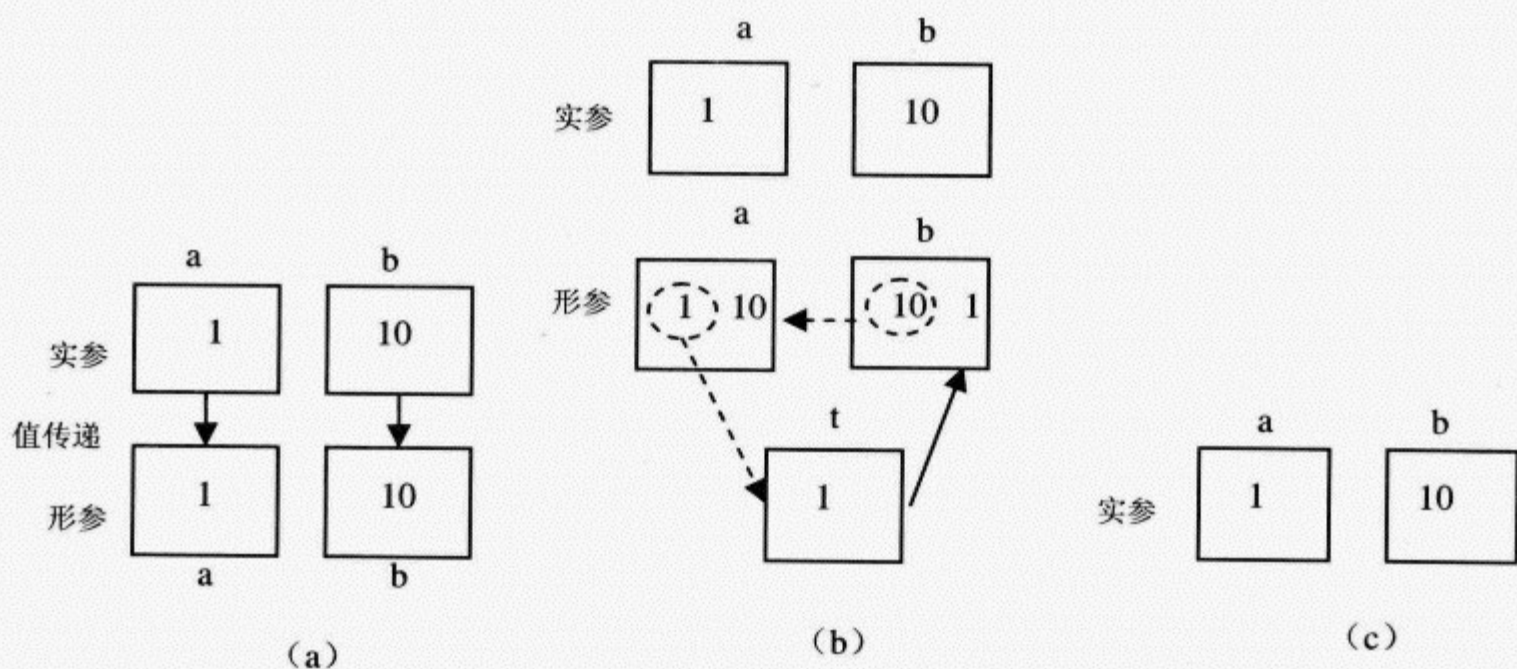


图 6-11 例程 6-8 的执行过程

图 6-10 中，(a) 为函数调用时实参与形参的数据传递过程，参数传递后形参中的值与实参保持一致；(b) 为执行函数时函数内部的对调操作，可以看出操作并不影响实参的值；(c) 为被调函数执行完毕后，所有的局部变量被释放掉，实参中的值没有任何变化。

所以，上面编写的函数 swap 起不到使两个变量的内容对调的功能。于是想到在主函数中直接实现两个变量的内容对调，不再为此单独编写一个函数。但这样既不符合题目的要求，又不符合结构化程序设计中提倡的将函数作为基本的功能模块。那么如何才能将“两个变量的内容对调”的功能编写成一个函数呢？这就需要将指针变量作为函数的参数。

### 6.7.2 指针变量的函数参数

函数的参数不仅可以是基本类型的变量（整型变量、字符型变量等），还可以是指针类型的变量。其作用是将一个变量的地址传递给被调函数。这样就可以在被调函数中通过指针（地址）操纵主调函数中的变量了。所以可以用指针变量作为函数的参数解决前面提出的“两个变量的内容对调”的问题。



例程 6-9 用指针变量作为函数的参数解决两个变量的内容对调问题。

```
#include <stdio.h>
int main(void)
{
    int a=1,b=10;
    printf("a=%d,b=%d\n",a,b);
    /*用 swap 函数实现两数的对调*/
    swap(&a,&b);
    printf("a=%d,b=%d\n",a,b);
    getchar();
    return 0;
}
int swap(int *x,int *y)
{
    int t;
    t=*x;
    *x=*y;
    *y=t;
}
```

这里要注意，使用参数为指针变量的函数时，函数调用的参数表列中参数一定是变量的地址，即实参一定是变量的地址：

```
swap(&a,&b);
```

被调函数定义的参数表列中参数一定是定义指针变量的形式，即形参一定是定义指针变量的形式：

```
void swap(int *a,int *b)
```

本程序中，将两个变量 a, b 的地址&a, &b 作为实参传递给形参。这样被调函数实际上接收到的是变量 a, b 的地址。那么在被调函数 swap 中，\*a, \*b 实际上就是对主调函数中的变量 a, b 的间接访问。对于本程序的执行过程可以用图 6-12 形象地展示出来。

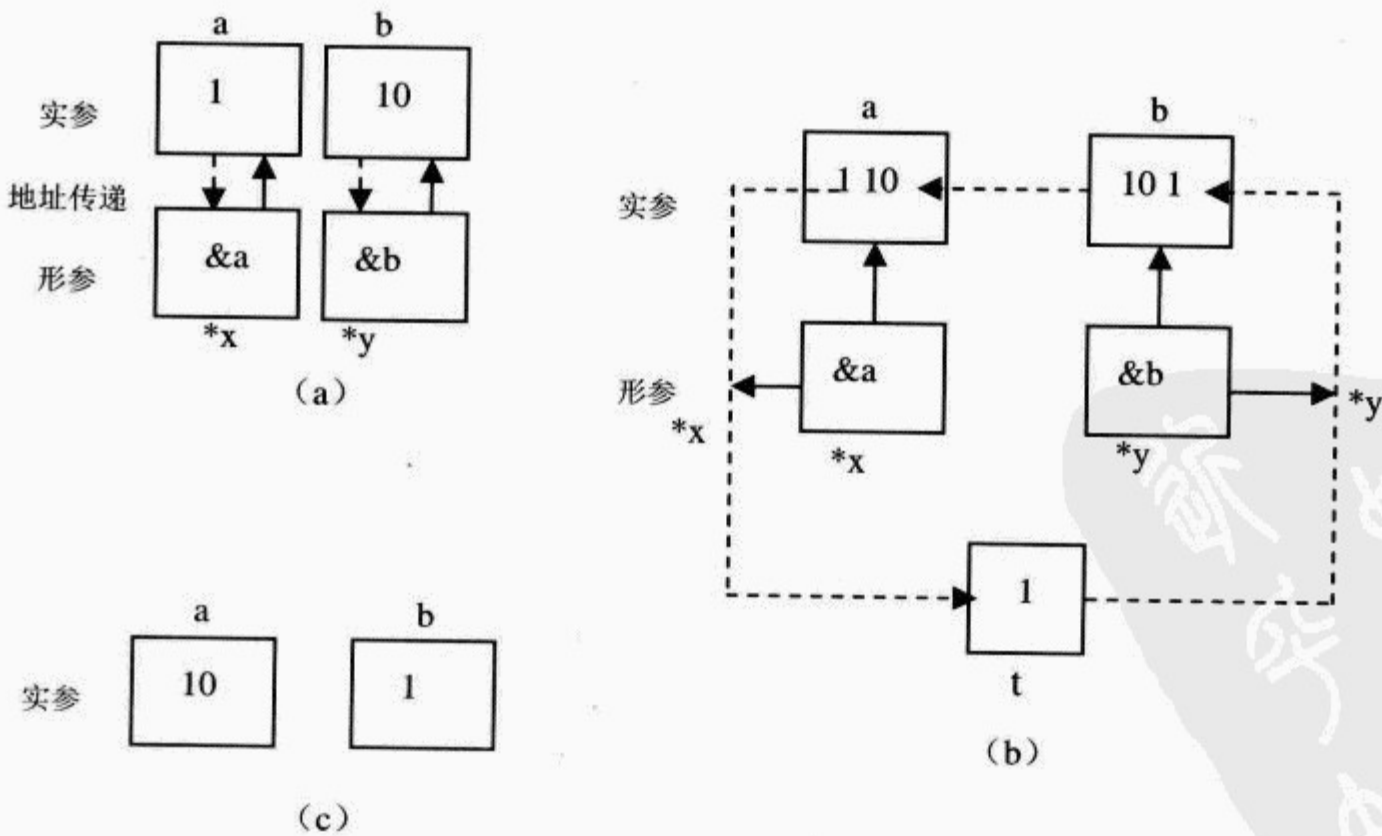


图 6-12 例程 6-9 的执行过程

图 6-12 中, (a) 表示实参将变量  $a, b$  的地址传递给形参  $x, y$ , 这样指针变量  $x, y$  就分别指向变量  $a, b$ 。(b) 表示在被调函数 `swap` 中, 通过  $*x, *y$  操作和中间变量  $t$  实现变量  $a, b$  的对调。首先程序将  $*x$  的值, 即  $*(&a)$  的值 1 赋值给中间变量  $t$ ; 然后将  $*y$  的值, 即  $*(&b)$  的值 10 赋值给  $*x, *(&a)$  也就是  $a$  本身; 最后将  $t$  中的值 1 赋值给  $*y, *(&b)$  也就是  $b$  本身。(c) 表示函数 `swap` 调用完毕后, 主调函数中的实参变量  $a, b$  实现了对调。

本例程的运行结果为:

```
a=1,b=10
a=10,b=1
```

其实, 将指针作为函数的参数传递给被调函数, 目的是使被调函数通过传递下来的变量的地址操纵主调函数中的变量。这样即便函数调用完毕后所有的局部变量都被释放掉, 也不会影响函数的效果。因为在函数执行过程中就已经对主调函数中的变量进行了操作。下面再通过一个例子来进一步理解指针变量作为函数参数的用法。

#### 例程 6-10 输入 $x, y, z$ 三个整数, 编写一个程序使它们按从小到大的顺序输出。

```
#include <stdio.h>
int main(void)
{
    int x,y,z;
    scanf("%d,%d,%d",&x,&y,&z);
    exchange(&x,&y,&z);
    printf("%d,%d,%d\n",x,y,z);
    getchar();
    return 0;
}
int exchange(int *p1,int *p2,int *p3)
{
    if(*p1>*p2)
        swap(p1,p2);
    if(*p2>*p3)
        swap(p2,p3);
    if(*p1>*p2)
        swap(p1,p2);
}
int swap(int *a,int *b)
{
    int t;
    t=*a;
    *a=*b;
    *b=t;
}
```

本例程通过指针变量作为函数参数实现将主调函数中的变量  $x, y, z$  按从小到大的顺序输出。具体的做法是: 将变量  $x, y, z$  的地址作为 `exchange` 函数的参数传递给函数 `exchange`。在 `exchange` 中通过变量  $x, y, z$  的指针对变量  $x, y, z$  的内容进行调整, 即将最小的数存放到变量  $x$  中, 将最大的数存放到变量  $z$  中。在函数 `exchange` 中调用了函数 `swap`, 它的作用是实现两个变量值的对调。本例程的运行结果为:

```
2,5,1
1,2,5
```

其实, 在本程序中真正起到通过指针对变量  $x, y, z$  的内容进行调整的函数是 `swap`,



函数 `exchange` 在这里只起到比较 `x`, `y`, `z` 内容的大小, 根据比较结果调用 `swap` 函数, 并继续向 `swap` 函数传递 `x`, `y`, `z` 的指针的作用。

## 6.8 指向数组元素的指针

前面已经讲过, 数组是同种类型变量的集合。既然任何一个变量都有它的指针(地址), 那么每个数组元素也都有它的指针, 因为每个数组元素在内存中都有自己的存储空间。

数组元素有其指定的类型, 整型数组中每个元素都是整型; 字符型数组中每个元素都是字符型。因此, 定义指向数组元素的指针变量的方法与定义指向变量的指针变量的方法相同。

例如:

```
int a[10];  
int *p, *q;
```

`p`, `q` 既可以理解为指向整型变量的指针变量, 也可以理解为指向整型数组元素的指针变量。因此 `p`, `q` 只要通过下面的赋值语句就可以指向整型数组中的每一个元素。

```
p=&a[0]; /*把 a[0] 元素的地址赋值给 p*/  
q=&a[9]; /*把 a[9] 元素的地址赋值给 q*/
```

如图 6-13 所示。

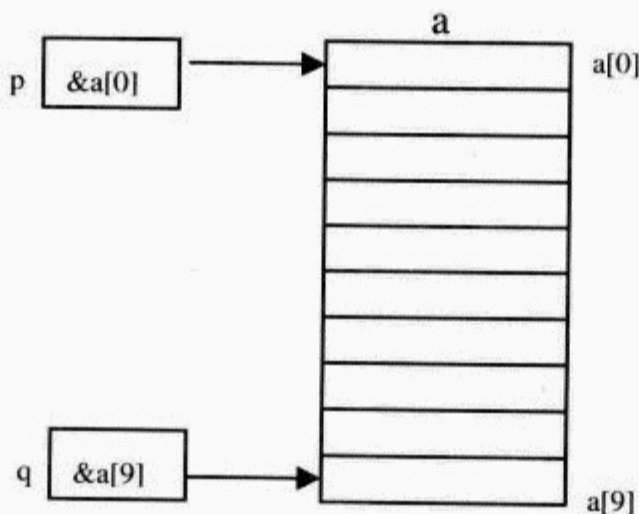


图 6-13 指向数组元素的指针 `p`, `q`

指针变量 `p` 中存放着 `a[0]` 的地址, 因此 `p` 指向 `a[0]`; 指针变量 `q` 中存放着 `a[9]` 的地址, 因此 `q` 指向 `a[9]`。

C 语言规定, 数组名代表数组的首地址, 也就是第 0 号元素的地址。因此, 下面两个语句等价:

```
p=&a[0];  
p=a;
```

在定义指向数组元素的指针变量时同样可以进行初始化。

例如:

```
int a[10];
```

```
int *p=&a[10];
```

等价于：

```
int a[10],*p;  
p=&a[10];
```

也等价于：

```
int a[10];  
int *p=a;
```

或者

```
int a[10],*p;  
p=a;
```

## 6.9 用指针引用数组元素

可以通过数组名加下标的方法对数组元素引用。例如：

```
x=a[5];
```

将数组元素 `a[5]` 赋值给变量 `x`。也可以通过指针的方法引用数组元素。

### 6.9.1 指针对数组元素的引用

6.8 节中已经讲到，每个数组元素都有其地址，因此通过指针引用数组元素与通过指针引用变量的方法是一样的。例如：`p` 定义为一个指向整型变量的指针变量，且 `p` 指向一个整型的数组元素，则可以通过 `p` 引用数组中的元素。

```
a=*p;
```

表示把 `p` 指向的数组元素赋值给变量 `a`。

```
*p=1;
```

表示对 `p` 指向的数组元素空间赋值 1。

数组是一种线性数据结构，即在内存中是一段连续的存储空间。因此，数组中的数组元素的地址也应该是连续的。在 C 语言中规定，如果指针变量 `p` 指向数组中的一个元素，则 `p+1` 就指向该元素的下一个元素。这里必须注意，下一个元素的地址未必就是简单的 `p+1`，如图 6-14 所示。

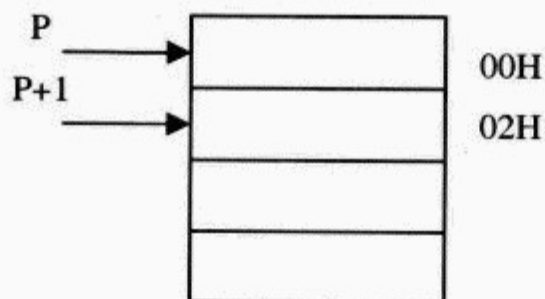


图 6-14 实际地址与指针



第一个存储单元的单元地址为 00H，第二个存储单元的单元地址为 02H，并不是简单的加 1 关系（这可能是因为这是一个整型数组，每个数组元素都是整型数，占 2 字节，系统按字节编址）。但是如果已知 p 指向 00H 单元，则 p+1 就指向该元素的下一个元素，即 02H 单元。编译系统会将 p+1 理解为 p+1\*d，其中 d 是一个元素所占的字节数。

这样只要知道了数组元素的指针，就可以通过加 1 减 1 操作找到其他数组元素的地址，从而访问到其他的数组元素。一般地，数组的第一个元素的地址（首地址）是知道的，因为这个地址可以用该数组的数组名表示。因此，只要知道了一个数组的数组名，就可以通过指针的运算访问到该数组的所有数据元素。

前面讲过，引用一个数组元素的方法是通过数组的下标和数组名实现的。例如 a[5]就是指数组 a 的第 6 个元素。现在又有一种方法来引用一个数组元素了，就是通过数组名的加 1 减 1 操作找到要引用的元素的地址，然后取值。例如 a[5]就等价于\*(a+5)，即先将数组的首地址加 5，找到第 6 个元素的地址，然后进行取值“\*”运算，得到数组第 6 个元素的值。其实“[]”是一种变址运算符，a[i]就相当于先按 a+i 计算地址，然后找出该地址单元中的值。即 a[i]等价于\*(a+i)。

综上所述，引用一个数组元素可以用以下两种方法：

- (1) 下标法，即用 a[i]形式访问数组元素。前面介绍的数组都是采用这种方法。
- (2) 指针法，即采用\*(a+i)或\*(p+i)形式间接访问数组元素，其中 a 是数组名，p 是指向数组的指针变量。

下面通过例子来理解用指针引用数组元素。

#### 例程 6-11 用指针引用数组元素的方法实现选择排序（从小到大排序）。

分析：例程 6-2 已经详细介绍了选择排序。这里依然用数组存放待排序数列，但要求用指针引用数组元素的方法实现对数组元素的重新排序。下面给出程序代码：

```
#include <stdio.h>
#define MAX 10
int main(void)
{
    int i, j, t, sort[MAX], min;
    printf("Please input ten integer:\n");
    for(i=0; i<MAX; i++)
        scanf("%d", sort+i);
    for(i=0; i<MAX-1; i++)
    {
        min=*(sort+i);
        for(j=i+1; j<MAX; j++)
            if(*(sort+j)<min)
            {
                t=min;
                min=*(sort+j);
                *(sort+j)=t;
            }
        *(sort+i)=min;
    }
    printf("The result of sort is:\n");
    for(i=0; i<MAX; i++)
        printf("%d ", *(sort+i));
}
```

```
    getchar();  
    return 0;  
}
```

实际上只要将例程 6-2 代码中的 `sort[i]` 和 `sort[j]` 分别改为 `*(sort+i)` 和 `*(sort+j)` 就是本代码了。`sort` 是待排序数列的首地址, `(sort+i)` 就是由首单元向后第 `i` 个存储单元的地址。`sort[i]` 等价于 `*(sort+i)`。

值得注意的是, 在 `scanf("%d", sort+i)` 一句中, 并没有把原来的 `scanf("%d", &sort[i])` 中的 `&sort[i]` 直接改为 `&*(sort+i)`, 而是改成了 `sort+i`。因为 `&sort[i]` 的作用是取 `sort[i]` 元素的地址, 而它的地址就是在首地址 `sort` 向后的第 `i` 个存储单元的地址, 即 `sort+i`。因此, `&sort[i]`, `&*(sort+i)`, `sort+i` 三者是等价的。

本例程的运行结果为:

```
Please input ten integer:  
0 9 8 7 6 5 4 3 2 1  
The result of sort is:  
0 1 2 3 4 5 6 7 8 9
```

## 6.9.2 几点注意事项

在使用指针来引用数组元素时, 要注意以下几点。

### (1) 注意指针变量和指针常量

先看下面一个例子。

#### 例程 6-12 用指针引用数组元素的方法输出一个数组中的元素。

```
#include <stdio.h>  
int main(void)  
{  
    int a[5], i, *p;  
    p=a;  
    for(i=0; i<5; i++)  
        scanf("%d", p++);  
    p=a;  
    for(i=0; i<5; i++)  
        printf("%d ", *(p++));  
    getchar();  
    return 0;  
}
```

本例程中完全应用指针对数组元素进行操作。首先定义一个指向整型变量的指针变量 `p`, 将 `p` 指向数组 `a` 的首单元, 即 `p=a`, 然后通过指针变量 `p` 向数组输入数据。`p++` 表示每次循环指针变量 `p` 自动加 1, 也就是自动指向下一个数据元素单元。接下来将 `p` 重新指向数组 `a` 的首单元, 因为最后 `p` 已指向数组的最后一个元素的下一个单元, 然后循环输出该数组中的数据元素。本程序的运行结果为:

```
1 2 3 4 5  
1 2 3 4 5
```

既然 `p` 中的内容就是 `a` 中的内容, 那么可否将源程序中的 `p++` 改为 `a++` 呢? 答案是不能的。这就牵扯到指针变量和指针常量的问题。`a` 作为数组名虽然表示数组的首地址, 但它是一个指针常量, 是一个固定的地址而不能被改动。`a++` 实际上是执行自增 1 操作, 它相当



于  $a=a+1$ ，所以对指针常量进行自增 1 操作是错误的，但是  $p++$  是可以的，因为  $p$  是先前定义的指针变量。在编程时一定要注意这一点。

### (2) 注意指针变量运算中的运算符

- ✧  $*p++$  的执行顺序是自右而左的，它等价于  $*(p++)$ 。
- ✧  $*(p++)$  与  $*(++p)$  作用不同。前者是先计算  $*p$ ，再进行  $p++$  操作；后者是先进行  $p++$  操作，再计算  $*p$ 。若  $p$  的初值为  $a$ ，则  $*(p++)$  等价  $a[0]$ ， $*(++p)$  等价  $a[1]$ 。
- ✧  $(*p)++$  表示  $p$  所指向的元素值加 1。
- ✧ 如果  $p$  当前指向  $a$  数组中的第  $i$  个元素，则： $*(p-)$  相当于  $a[i-]$ ； $*(++p)$  相当于  $a[++i]$ ； $*(-p)$  相当于  $a[-i]$ 。

### (3) 注意指针的指向

由于可以通过指针来引用数据元素，因此在编写程序时就要控制好指针的指向，否则就会出现意想不到的错误。例如将例程 6-12 代码中的第二条  $p=a$  忽略掉，就会出现这样的运行结果：

```
1 2 3 4 5
-26 292 1370 1 -28
```

这是因为执行完循环输入后， $p$  已指向数组的最后一个元素的下一个单元。因此这时必须将  $p$  重新指向数组  $a$  的首单元，否则再读的就是数组  $a$  以外的元素了。

## 6.10 数组名作为函数的参数

数组名的本质是数组的首地址，因此可以将数组名作为函数的参数传递给被调函数，这样就可以在被调函数中对主调函数中定义的数组进行访问和操作了。

### 6.10.1 数组名参数

数组名作为函数的参数与指针作为函数的参数本质是一样的。都是通过主调函数中的变量或数组元素的地址对它们进行操作。

调用数组名作为参数的函数的方法如下。

```
main()
{int array[10];
    ....
    ....
    f(array,10);
    ....
    ....
}

f(int arr[],int n);
{
    ....
    ....
}
```

主调函数 main 中调用函数 f, f 有两个实参, 一个为数组 array 的数组名, 即首地址; 一个为数组 array 的长度。在函数 f 的定义中, f 有两个形参, 一个为写成数组形式的 arr[ ]; 一个为 n, 用来接收实参传递下来的数组长度。其中参数 int arr[ ] 相当于定义了一个指针变量 int \*arr, 用来接收实参传递下来的数组首地址。因此, 上面函数 f 的定义也可以写成:

```
f(int *arr,int n);
{
    ....
    ....
}
```

二者的作用是一样的。

下面通过一个例子来理解数组名作为函数参数的用法。

### 例程 6-13 数组名作为函数的参数。

```
#include <stdio.h>
int main(void)
{
    int array[5];
    InputArr(array,5);
    OutputArr(array,5);
    getchar();
    return 0;
}
InputArr(int *arr,int n)
{
    int i;
    for(i=0;i<n;i++)
        scanf("%d", (arr+i));
}
OutputArr(int *arr,int n)
{
    int i;
    for(i=0;i<n;i++)
        printf("%d ",*(arr+i));
}
```

本例程在主函数中定义了数组 array[5], 然后通过函数 InputArr 向该数组输入数据, 再通过函数 OutputArr 读取该数组中的数据, 这里面用到了数组名作为函数的参数。在调用函数 InputArr 和 OutputArr 时, 将数组名 array 作为实参传递给形参 arr, 这样在被调函数中就可以通过对数组元素地址的控制来访问主函数中定义的数组。即便在函数 InputArr 和 OutputArr 调用完毕后所有被调函数中的局部变量都被释放掉, 也不会影响数组 array 的状态, 因为在函数 InputArr 和 OutputArr 执行过程中就已经访问了数组 array。程序的整个执行过程如图 6-15 所示。



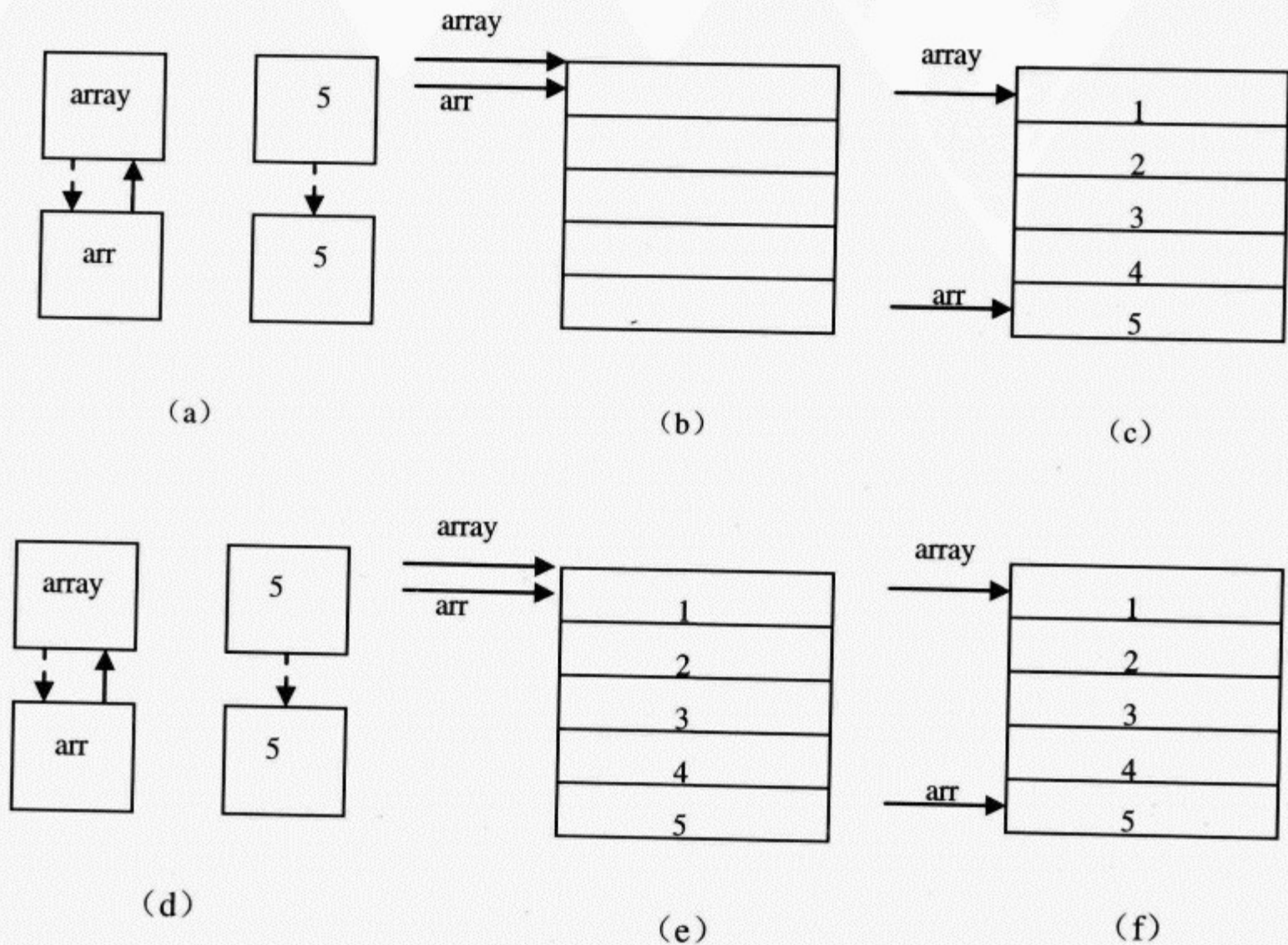


图 6-15 例程 6-13 执行过程

图 6-14 中，(a) 表示调用函数 `InputArr` 时的参数传递。将数组 `array` 的数组名作为实参传递给形参 `arr`，`arr` 是一个定义的指针变量，可以存放一个指针。这样指针变量 `arr` 就指向了数组的首单元。(b) 表示 `array` 和 `arr` 同时指向数组的首单元。这里要注意 `array` 就是数组的首地址，它是指针常量；而 `arr` 是被调函数 `InputArr` 中的指针变量，它接收实参传递下来的数组首地址 `array`，因此也指向数组 `array` 的首单元。(c) 表示函数 `InputArr` 中的操作，即向数组中输入数据。这里用到一个循环语句：

```
for(i=0;i<n;i++)
    scanf("%d", (arr+i));
```

其中参数 `(arr+i)` 为当前要输入数据的数组元素的地址。随着 `i` 的增加，指针不断向数组尾部移动，但是指针 `array` 始终不变，因为它是指针常量。(d) 表示调用函数 `OutputArr` 的参数传递。(e) 表示 `array` 和 `arr` 同时指向数组的首单元，但这时数组 `array` 中已存满了刚才输入的数据。(f) 表示函数 `OutputArr` 中的操作，即循环输出数组中的元素。

本例程的运行结果为：

```
1 2 3 4 5
1 2 3 4 5
```

从这个例子中不难发现，将数组名作为函数的参数，可以在被调函数中控制主调函数中的数组，从而不必将这种控制数组的操作再写到主函数中。例如上面这个例子，常用的



方法是将对数组的输入输出等操作直接写到主函数中，这样固然可以，但是程序的结构就远不如将这些操作编写成一个一个的函数，再由主函数 main 来调用来得好。结构化程序设计一个最为突出的特点就是调用函数，即将特定功能的程序段编写成函数，通过函数的调用实现整个程序的功能。利用将数组名作为函数参数的方法，还可以将前几节给出的数列排序改进成函数调用的方法。

## 6.10.2 应用举例

**例程 6-14** 编写一个函数，实现“冒泡排序”，并给出测试函数。

```
#include "stdio.h"
void InputArr(int *,int );
void OutputArr(int *,int );
void BubbleSort(int *,int );
int main()
{
    int sort[5];
    printf("Please input 5 integer:\n");
    /*调用函数 InputArr 输入数组元素*/
    InputArr(sort,5);
    /*调用函数 BubbleSort 进行冒泡排序*/
    BubbleSort(sort,5);
    printf("The sort result is\n");
    /*调用函数 OutputArr 输出数组元素*/
    OutputArr(sort,5);
}
void InputArr(int *arr,int n)
{
    int i;
    for(i=0;i<n;i++)
        scanf("%d", (arr+i));
}
void OutputArr(int *arr,int n)
{
    int i;
    for(i=0;i<n;i++)
        printf("%d ",*(arr+i));
}
void BubbleSort(int *sort,int n)
{
    int i,j,t;
    for(i=0;i<n-1;i++) {
        for(j=0;j<n-1-i;j++)
            if(*(sort+j)>*(sort+j+1))
            {t=*(sort+j);
              *(sort+j)=*(sort+j+1);
              *(sort+j+1)=t;
            }
    }
}
```

本例程中，将排序序列的输入、排序、输出都独立编写成函数，由主函数 main 调用。函数之间的参数传递采用指针传递，数组第一个元素的内存地址传递给了函数。另外，为了知道数组的大小，每个函数都把数组的大小作为其参数之一。

将这些操作独立编写成函数的好处是：程序的结构更好、可读性更强、软件的可重用性更好。因此在编程时，建议将实现特定功能的程序段编写成函数。



数组名作为函数的参数，本质上就是 6.2.4 节中讲到的指针（或指针变量）作为函数的参数。它只是一个特例而已。采用指针作为函数的参数，不仅可以将对数组的操作放到独立的函数中进行，增强程序的结构性，还可以解决用一般的变量作为函数的参数解决不了的问题。例如它可以解决一个函数只能有一个返回值的瓶颈，看下面这个例子。

**例程 6-15** 编写一个函数，实现从 10 个数中找出最大值和最小值。

**分析：**这个题目要求函数能带回两个值，但是 C 语言中函数只能有一个返回值。因此想到了应用第 4 章讲到的全局变量在函数之间传递数据，实现“多返回值”。代码如下：

```
#include "stdio.h"
int max,min; /*全局变量*/
void max_min_value(int array[],int n)
{
    int *p,*array_end;
    array_end=array+n;
    max=min=*array;
    for(p=array+1;p<array_end;p++)
        if(*p>max)max=*p;
        else if (*p<min)min=*p;
    return;
}
int main()
{
    int i,number[10];
    printf("Input 10 integer numbers:\n");
    for(i=0;i<10;i++)
        scanf("%d",&number[i]);
    max_min_value(number,10);
    printf("\nmax=%d,min=%d\n",max,min);
    getch();
    return 1;
}
```

本程序中，定义了两个全局变量：min 和 max，函数 max\_min\_value 中求出的最大值和最小值放在 max 和 min 中。由于它们是全球变量，因此在主函数中也可以直接使用。函数 max\_min\_value 中，在执行 for 循环时，p 的初值为 array+1，也就是使 p 指向 array[1]。以后每次执行 p++，使 p 指向下一个元素。每次将 \*p 和 max 与 min 比较，将大者放入 max，小者放入 min。这是一种获得一个数列中最大值和最小值的经典算法。

但是这样的解决方案也并不是很好，因为用到了全局变量会破坏整个程序的结构，破坏了程序设计中最低访问权限的原则。

其实在这里不仅可以将数组名作为函数的参数来传递，以实现数组的独立操作，还可以将全局变量 max 和 min 定义为主函数中的局部变量，将 max 和 min 的地址作为函数的参数传递给被调函数 max\_min\_value，同样可以实现“多返回值”。下面给出改进后的程序代码：

```
#include "stdio.h"
void max_min_value(int array[],int n,int *min,int *max)
{
    int *p,*array_end;
    array_end=array+n;
    *max=*min=*array;
    for(p=array+1;p<array_end;p++)
```



```
        if(*p>*max)*max=*p;
        else if (*p<*min)*min=*p;
    return;
}
int main()
{
    int i,number[10];
    /*定义变量min和max,用来存放最大值和最小值*/
    int min,max;
    printf("Input 10 integer numbers:\n");
    for(i=0;i<10;i++)
        scanf("%d",&number[i]);
    /*函数max_min_value增加了两个指针参数*/
    max_min_value(number,10,&min,&max);
    printf("\nmax=%d,min=%d\n",max,min);
    getch();
    return 1;
}
```

本程序在主函数中定义了两个局部变量 min 和 max,用来存放 10 个数中的最小值和最大值。在函数 max\_min\_value 中增加了两个参数,分别传递变量 min 和 max 的地址。这样在被调函数 max\_min\_value 中,就可以通过 min 和 max 的地址改变它们的取值,而不必再定义全局变量了。本例程的运行结果为:

```
Input 10 integer numbers:
1 2 3 4 5 6 7 8 9 10
max=10,min=1
```

数组名作为函数的参数,可以将对数组的操作放到独立的函数中进行,使程序的结构性更好,可读性更强。数组名的本质是数组的首地址,通过理解数组名作为函数的参数,加深对指针作为函数的参数的理解。

## 6.11 二维数组的指针

指针变量既可以指向一维数组中的元素,也可以指向多维数组中的元素。本小节以二维数组为例介绍二维数组的指针。

### 6.11.1 二维数组的地址

为了更加清楚地解释二维数组的地址,通过一个实例来说明。设有整型二维数组 a[3][4] 如下:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

在 C 语言中可以定义为:

```
int a[3][4]={ {0,1,2,3},{4,5,6,7},{8,9,10,11}}
```

假设数组 a 的首地址为 1000,各下标变量的首地址及其值如图 6-16 所示。



1000 0	1002 1	1004 2	1006 3
1008 4	1010 5	1012 6	1014 7
1016 8	1018 9	1020 11	1022 12

图 6-16  数组 a 的内部存储形式

因为是整型数组，所以每个存储单元占两字节。又因为系统按字节编址，所以每个数组元素的地址都是偶数。

前面已经介绍过，C 语言允许把一个二维数组分解为多个一维数组来处理。因此数组 a 可分解为 3 个一维数组，即 a[0]，a[1]，a[2]。每一个一维数组又含有 4 个元素，如图 6-17 所示。

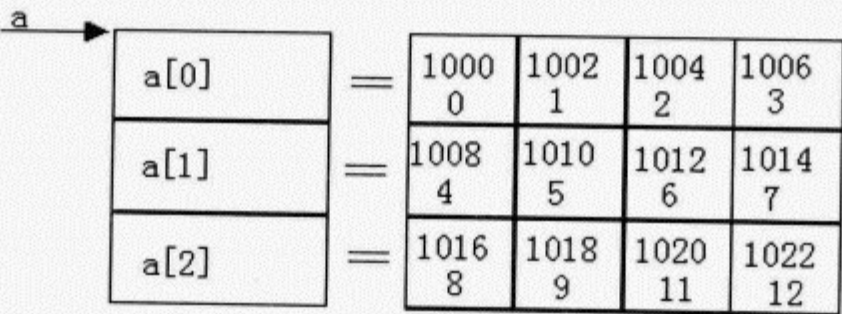


图 6-17  将二维数组看成一维数组

这样数组 a 就可以看成一个一维数组 a[3]，共有 3 个数组元素 a[0]、a[1]、a[2]。而每一个数组元素又包含一个一维的数组。例如 a[0]数组，含有 a[0][0]、a[0][1]、a[0][2]、a[0][3]4 个元素。

那么 a 自然可以看成一维数组 a[3]的首地址，即 a[0]的地址，等于 1000。a+1 就可以看成是 a[1]的地址，等于 1008。若从二维数组的角度来看，a 是二维数组名，a 代表整个二维数组的首地址，也是二维数组 0 行的首地址，即等于 1000。a+1 代表第一行的首地址，等于 1008，如图 6-18 所示。

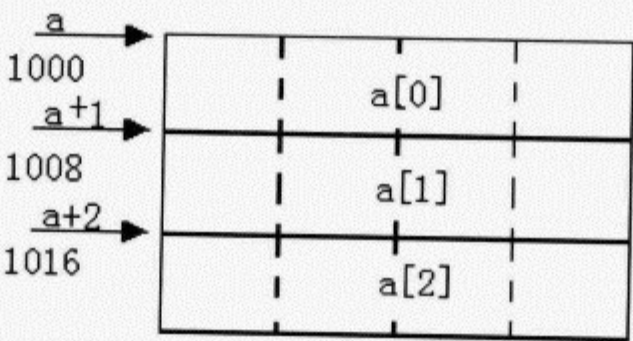


图 6-18  二维数组每行的首地址

前面已经讲过，\*(a+0)或\*a 是与 a[0]等效的，它表示一维数组 a[0]的 0 号元素的首地址，因此也就是 1000。&a[0][0]是二维数组 a 的 0 行 0 列元素首地址，同样是 1000。所以，a，a[0]，\*(a+0)，\*a，&a[0][0]是等同的。

同理，a+1 是二维数组 1 行的首地址，等于 1008。a[1]也是一维数组 a[3]的第二个元素，可以看成是二维数组 a[3][4]第二行的首地址，因此也为 1008。&a[1][0]是二维数组 a 的 1

行 0 列元素地址，也是 1008。因此  $a+1$ ， $a[1]$ ， $*(a+1)$ ， $\&a[1][0]$ 是等同的。

由此可得出： $a+i$ ， $a[i]$ ， $*(a+i)$ ， $\&a[i][0]$ 是等价的。

需要说明一点的是， $\&a[i]$ 和  $a[i]$ 也是等同的。在二维数组中不能将 $\&a[i]$ 理解为元素  $a[i]$ 的地址，因为根本就不存在元素  $a[i]$ ，它只是一种地址计算方法，表示二维数组  $a$  第  $i$  行首地址。因此也可以得出： $a[i]$ ， $\&a[i]$ ， $*(a+i)$ 和  $a+i$  都是等价的。

另外， $a[0]$ 也可以看成是一维数组  $a[0]$ （二维数组  $a[3][4]$ 的第一行）的第 0 号元素的首地址，而  $a[0]+1$  则是  $a[0]$ 的 1 号元素首地址，由此可得出  $a[i]+j$  则是一维数组  $a[i]$ 的  $j$  号元素首地址，它等于 $\&a[i][j]$ ，如图 6-19 所示。

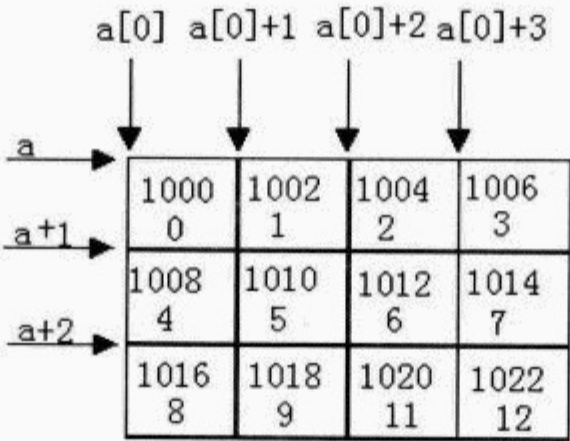


图 6-19 二维数组每行中每列的地址

由  $a[i]=*(a+i)$ 得  $a[i]+j=*(a+i)+j$ 。由于 $*(a+i)+j$  是二维数组  $a$  的第  $i$  行第  $j$  列元素的地址，所以，该元素的值等于 $*(*(a+i)+j)$ 。 $*(*(a+i)+j)=a[i][j]$ 。

这里必须注意，不要把  $a[i]+j$  与  $a+j$  混淆。 $a[i]+j$  表示二维数组  $a[3][4]$ 中第  $i$  行第  $j$  列的地址；而  $a+j$  表示二维数组  $a[3][4]$ 第  $j$  行的首地址。

下面通过一个例子来对上面所讲到的概念加以总结。

例程 6-16 二维数组中的相关概念。

```
#include "stdio.h"
int main(){
    int a[3][4]={0,1,2,3},{4,5,6,7},{8,9,10,11}};
    /*输出二维数组 a[3][4] 第一行的首地址*/
    printf("%d",a);
    printf("%d",*a);
    printf("%d",a[0]);
    printf("%d",&a[0]);
    printf("%d\n",&a[0][0]);
    /*输出二维数组 a[3][4] 第二行的首地址*/
    printf("%d",a+1);
    printf("%d",*(a+1));
    printf("%d",a[1]);
    printf("%d",&a[1]);
    printf("%d\n",&a[1][0]);
    /*输出二维数组 a[3][4] 第三行的首地址*/
    printf("%d",a+2);
    printf("%d",*(a+2));
    printf("%d",a[2]);
    printf("%d",&a[2]);
    printf("%d\n",&a[2][0]);
    /*输出二维数组 a[3][4] 第二行第二列的地址*/
```



```

printf("%d", a[1]+1);
printf("%d", &a[1][1]);
printf("%d\n", *(a+1)+1);
/*输出二维数组 a[3][4] 第二行第二列的元素值*/
printf("%d", *(a[1]+1));
printf("%d", *(*a+1)+1);
printf("%d\n", a[1][1]);
getch();
return 1;
}

```

通过本例程可以总结一下上面所讲到的概念。对于二维数组  $a[3][4]$ ,

- (1)  $a$ ,  $*a$ ,  $a[0]$ ,  $*(a+0)$ ,  $\&a[0]$ ,  $\&a[0][0]$  都可以表示数组第一行的首地址。
- (2)  $a+1$ ,  $a[1]$ ,  $*(a+1)$ ,  $\&a[1]$ ,  $\&a[1][0]$  都可以表示数组第二行的首地址。
- (3)  $a+2$ ,  $a[2]$ ,  $*(a+2)$ ,  $\&a[2]$ ,  $\&a[2][0]$  都可以表示数组第三行的首地址。
- (4)  $a[1]+1$ ,  $*(a+1)+1$ ,  $\&a[1][1]$  都可以表示数组第二行第二列的地址。
- (5)  $*(a[1]+1)$ ,  $*(*(a+1)+1)$ ,  $a[1][1]$  都可以表示数组第二行第二列的元素值。

从而可以得到这样一个普遍的结论:

- (1)  $a+i$ ,  $a[i]$ ,  $*(a+i)$ ,  $\&a[i]$ ,  $\&a[i][0]$  都可以表示数组第  $i$  行的首地址。
- (2)  $a[i]+j$ ,  $*(a+i)+j$ ,  $\&a[i][j]$  都可以表示数组第  $i$  行第  $j$  列的地址。
- (3)  $*(a[i]+j)$ ,  $*(*(a+i)+j)$ ,  $a[i][j]$  都可以表示数组第  $i$  行第  $j$  列的元素值。

因此本例程的运行结果为:

```

-64, -64, -64, -64, -64
-56, -56, -56, -56, -56
-48, -48, -48, -48, -48
-54, -54, -54
5, 5, 5

```

其中 -64, -56, -48, -54 都表示内存地址, 没有实际意义。5 为  $a[1][1]$  (第二行第二列) 的值。

### 6.11.2 指向二维数组的指针

把二维数组  $a$  按行分解为一维数组  $a[0]$ 、 $a[1]$ 、 $a[2]$  之后, 设  $p$  为指向二维数组的指针变量。可定义为:

```
int (*p)[4]
```

它表示  $p$  是一个指针变量, 指向包含 4 个元素的一维数组, 即二维数组的每一行。 $p+i$  就表示指向二维数组的第  $i+1$  行。例如:  $p+1$  就表示指向二维数组第二行的指针。从前面的分析可得出  $*(p+i)+j$  是二维数组  $i$  行  $j$  列元素的地址, 而  $*(*(p+i)+j)$  则是  $i$  行  $j$  列元素的值。

二维数组指针变量说明的一般形式为:

```
类型说明符 (*指针变量名)[长度]
```

其中“类型说明符”为所指向数组的数据类型, “\*”表示其后的变量是指针类型, “长度”表示二维数组分解为多个一维数组时一维数组的长度, 也就是二维数组的列数。应注

意“(\*指针变量名)”两边的括号不可少,否则表示的意义就不同了。

下面通过例子来理解指向二维数组的指针变量。

### 例程 6-17 编写一个函数实现矩阵的转置运算。

分析:要想在被调函数中操纵主调函数中的二维数组,可以用指向数组的指针作为函数的参数,这样可以使程序的结构更好。代码如下:

```
#include <stdio.h>
void InputMatrix(int (*a)[4],int ,int );
void OutputMatrix(int (*b)[3],int ,int );
void reverse(int (*a)[4],int (*b)[3]);
int main(void)
{
    int a[3][4],b[4][3];
    printf("Please input 3X4 matrix\n");
    InputMatrix(a,3,4);
    reverse(a,b);
    printf("The revier matrix is\n");
    OutputMatrix(b,4,3);
    getchar();
    return 0;
}
void InputMatrix(int (*a)[4],int n,int m)
{
    /*输入n*m阶的矩阵*/
    int i,j;
    for(i=0;i<n;i++)
        for(j=0;j<m;j++)
            scanf("%d",&*(a+i)+j);
}
void OutputMatrix(int (*b)[3],int n,int m)
{
    /* 输出n*m阶矩阵的值*/
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
            printf("%d ",*(*(b+i)+j));
        printf("\n");
    }
}
void reverse(int (*a)[4],int (*b)[3])
{
    /*矩阵的转置运算*/
    int i,j;
    for(i=0;i<4;i++)
        for(j=0;j<3;j++)
            b[i][j]=a[j][i];
}
```

本程序中,函数 InputMatrix 用来向二维数组 a[3][4]中输入数据;函数 reverse 用来实现矩阵的转置运算,将转置后的矩阵存入数组 b[4][3]中;函数 OutputMatrix 的作用是输出转置矩阵 b[4][3]。

在函数 InputMatrix 中,形参 a 被声明为指向一个包含 4 个元素的一维数组的指针变量。实参传递给形参二维数组 a[3][4]的第一行首地址。函数 InputMatrix 中的\*(a+i)+j 实际上是



指数组 `a[3][4]` 的第 `i` 行第 `j` 列的地址。函数 `InputMatrix` 通过这个地址向矩阵 `a[3][4]` 中输入数据。

在函数 `reverse` 中, 形参 `a` 被声明为指向一个包含 4 个元素的一维数组的指针变量, 形参 `b` 被声明为指向一个包含 3 个元素的一维数组的指针变量。函数中的 `a[j][i]` 和 `b[i][j]` 实际上等价于 `*(a+j)+i` 和 `*(b+i)+j`, 分别是指数组 `a[3][4]` 的第 `j` 行第 `i` 列的元素和数组 `b[4][3]` 中第 `i` 行第 `j` 列的元素。通过这样循环地赋值, 将数组 `a` 中的元素转置后赋值给数组 `b`。

在函数 `OutputMatrix` 中, 形参 `b` 被声明为指向一个包含 3 个元素的一维数组的指针变量。实参传递给形参二维数组 `b[4][3]` 的第一行首地址。函数 `OutputMatrix` 中的 `*(b+i)+j` 实际上是指数组 `b[3][4]` 中第 `i` 行第 `j` 列的元素。这样通过循环语句输出数组 `b[3][4]` 中的元素。

本例程的运行结果为:

```
Please input 3x4 matrix
1 2 3 4
5 6 7 8
9 0 1 2
The revier matrix is
1 5 9
2 6 0
3 7 1
4 8 2
```

## 6.12 字符数组

除了前面介绍的整型数组、浮点型数组外, 还有一类非常重要的数组就是字符数组。用来存放字符量的数组称为字符数组。在操作上, 字符数组有一些与整型数组、浮点型数组不同的地方。所以本节单独介绍字符数组。

### 6.12.1 字符数组的定义和初始化

字符数组的定义与前面介绍的数值数组定义相同。

例如:

```
char c[10];
```

当然, 字符数组也可以是二维或多维数组。

例如:

```
char c[2][3];
```

即为二维字符数组。

与整型数组、浮点型数组一样, 字符数组也允许在定义时初始化赋值。

例如:

```
char c[10]={ 'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm'};
```

赋值后各元素的值在数组中的情况如图 6-20 所示。

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]
c		p	r	o	g	r	a	m	\0

图 6-20 赋值后各元素的值

可以注意到，c[9]并未被赋值，如果初值个数小于数组长度，则只将初值的字符赋值给数组中前面那些元素，其余的元素系统自动为之补'\0'。

如果对全体元素赋初值，可以省去长度说明。

例如：

```
char c[]={'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm'};
```

这时数组 c 的长度自动定为 9。字符串的长度与字符数组的长度是一样的。在赋初值的字符个数较多时，应用这种初始化字符数组的方法是很方便的。

当然也可以初始化一个二维数组。

例如：

```
char a[3][5]={{ '*', '*', '*', '*', '*'},{ 'H', 'E', 'L', 'L', 'O'},{ '*', '*', '*', '*', '*'}};
```

如果用下面这段循环语句输出该二维数组中的元素：

```
for(i=0;i<3;i++)
{
    for(j=0;j<5;j++)
        printf("%c",a[i][j]);
    printf("\n");
}
```

显示的结果为：

```
*****
HELLO
*****
```

6.12.2 字符串与字符串的结束标志

在 C 语言中没有专门的字符串变量，通常用一个字符数组来存放一个字符串。在定义字符数组时，可以将字符数组中的元素填满，即字符数组长度等于字符串的长度。

例如：

```
char str[5]={ 'a', 'b', 'c', 'd', 'e'};
```

字符串的长度为 5，字符数组的长度也为 5。但有时用字符数组的长度来判断字符串的长度显得过于死板。例如要输入学生的姓名，如果把学生的姓名作为字符串，长度当然不一样，因此就不好定义一个确定长度的字符数组来存放学生姓名。

C 语言中规定了字符串的结束标志为'\0'， 因此把一个字符串存入一个数组时，也可以把结束符'\0'存入数组，并以此作为该字符串是否结束的标志。在程序中，可以依靠'\0'的位置来判断字符串是否结束。这样就不必再用字符数组的长度来判断字符串的长度。系统定义的一些字符串处理库函数以及用于输入输出字符串的“%s”格式符都是以'\0'作为一个字



字符串结束标志的，并不需要用户控制。

C语言允许用字符串的方式对数组作初始化赋值。

例如：

```
char c[]={'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm'};
```

可写为：

```
char c[]={"c program"};
```

或去掉“{}”写为：

```
char c[]="c program";
```

用字符串方式赋值比用字符逐个赋值要多占一字节，用于存放字符串结束标志‘\0’，这是系统自动生成的，不用程序员自己添加。上面的数组c在内存中的实际存放情况如图6-21所示。

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]
c		p	r	o	g	r	a	m	\0

图 6-21 数组c在内存中的实际存放情况

可以看出，用字符串方式赋值的数组与逐个元素赋值的数组在内存中的存放情况是一样的。但是用字符串方式赋值更加简洁，而且在用字符串赋初值时一般无须指定数组的长度，系统会根据实际赋值的字符串长度自行处理（分配存储空间和添加结束标志‘\0’）。

### 6.12.3 字符数组的输入输出

字符数组的输入输出有两种方法。一种是与一般数值数组一样，采用逐个字符输入输出。另一种是一次性的输入输出，应用“%s”格式符。下面通过例子来理解这两种输入输出方法。

#### 例程 6-18 采用逐个字符输入输出的方法。

```
#include <stdio.h>
int main(void)
{
    char str[12];
    int i;
    for(i=0;i<12;i++)
        scanf("%c",&str[i]);
    for(i=0;i<12;i++)
        printf("%c",str[i]);
    getchar();
    return 0;
}
```

例程中定义的字符数组str长度为12，向数组中循环输入12个字符，然后再循环输出该字符串。本例程的运行结果为：

```
hello world!
hello world!
```

显然，这种输入输出字符串的方法显得很死板。因为事先程序员必须知道要输入的字符串的长度，才能设置字符数组的大小和循环输入的次数。因此用这种方法输入输出字符串并不实际。

#### 例程 6-19 采用一次性输入输出的方法。

```
#include <stdio.h>
int main(void)
{
    char str[20];
    scanf("%s",str);
    printf("%s",str);
    getchar();
    return 0;
}
```

应用“%s”格式符可以实现字符串的一次性输入输出。本程序有如下结果：

```
China
China
```

这是因为使用“%s”格式符输入字符串时，系统将输入的字符串自动存储到以 str 为数组名字符数组中。当键入回车或空格时，系统就自动认为该字符串输入完毕，如果数组 str 还有剩余的存储空间，系统自动添加'\0'作为字符串的结束标志，如图 6-22 所示。

str	C	h	i	n	a	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
-----	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

图 6-22 字符数组 str 的实际存放情况

str 中只有前 5 个单元存放了字符，其余是系统自动添加的'\0'作为字符串的结束标志。

输出字符串时仍然采用“%s”格式符。输出时系统只要遇到第一个'\0'就认为字符串到此结束，停止输出。

因此，当用 scanf 函数输入字符串时，字符串中不能含有空格，否则将以空格作为字符串的结束符，输出的结果自然也就与输入的字符串不同了。

例如输入字符串：

```
Hello world!
```

输出的结果为：

```
Hello
```

因为字符数组 str 中只存放了字符串"Hello"，以后都是结束标志'\0'。为了避免这种情况，可多设几个字符数组分段存放含空格的字符串。

程序可改写如下：

```
#include <stdio.h>
int main(void)
{
    char str1[10],str2[10],str3[10];
    scanf("%s%s%s",str1,str2,str3);
    printf("%s %s %s",str1,str2,str3);
}
```



```
    getchar();  
    return 0;  
}
```

这样就可以利用 `scanf` 函数输入多个字符串，每个字符串长度小于 10，字符串之间以空格分隔。本程序有如下结果：

```
I love China!  
I love China!
```

在使用“%s”格式符输入输出字符串时要注意以下几点：

(1) `scanf` 函数中输入项参数是字符数组的数组名，也就是字符数组的首地址。

```
scanf("%s",str);
```

其中，`str` 是数组名。

(2) `printf` 函数中输出项参数是字符数组的数组名，也就是字符数组的首地址。

```
printf("%s",str);
```

其中，`str` 是数组名。

(3) 实际输入的字符串长度不要大于字符数组的长度减 1，因为要保留一个存储单元，存放字符串结束标志。

采用一次性输入输出字符串的方法较逐个输入输出要好得多，因为程序员不用事先知道要输入的字符串的长度，这是软件设计的起码要求。虽然这种方法将空格作为字符串的结束标志，使得不能连续输入带空格的字符串，但是在实际的编程中，利用“%s”格式符一次性输入输出字符串的方法十分有用。

## 6.13 字符串指针

字符串是 C 语言中一个十分重要的概念。可以用字符数组来存储一个字符串，也可以仅用指针来指向一个字符串，通过指针对字符串进行管理。

### 6.13.1 用指针指向字符串

在字符数组中讲到可以用字符数组存放一个字符串。

例如：

```
char c[]="c program";
```

就是将字符串“c program”存入字符数组 `c` 中，这是字符数组初始化的一种方法。数组 `c` 的大小实际上是 10（由系统分配），最后一个存储单元存放的是字符串结束标志“\0”。所以用“%s”格式符输出该字符串为：

```
printf("%s",c);
```

输出的结果是：

```
c program
```

其在内存中的存放情况见图 6-20。

其实也可以用字符指针指向一个字符串，这样就不必定义字符数组了。

例如：

```
char *c="c program";
```

这里虽然没有定义字符数组来存放字符串，但系统是按照字符数组处理的。系统会在内存中开辟一段连续的存储空间（字符数组）来存放该字符串。对字符指针变量 *c* 的初始化，实际上是把字符串 "c program" 的第 1 个元素的地址，也就是内存中开辟的连续存储空间的首地址赋值给 *c*，如图 6-23 所示。

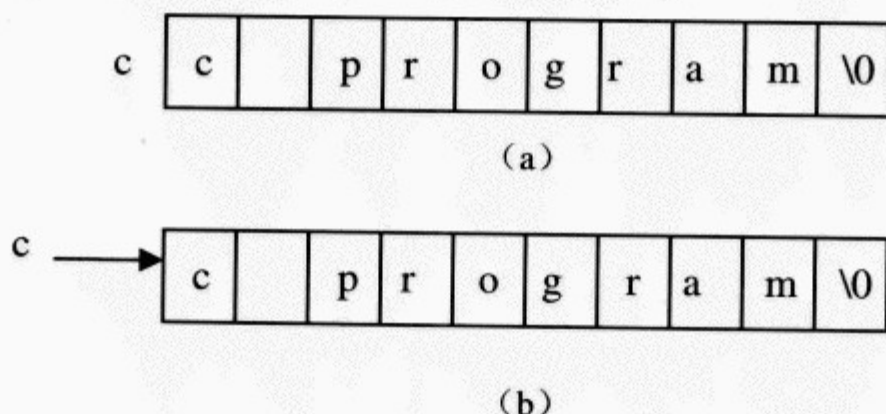


图 6-23 字符串数组和指针指向的字符串

图 6-23 中，(a) 表示字符数组 *c* 的内部存储情况。这里 *c* 是字符数组的数组名，也是字符数组的首地址，不过它只是一个地址常量。(b) 表示字符指针指向的字符串，*c* 是一个指针变量，存放了该字符串所在的字符数组（或者说是内存空间）的第一个元素的地址，也就是说指针变量 *c* 指向了该字符数组的第一个元素。

这两种字符串表示形式最后一个存储单元存储的内容都是 '\0'，这是系统自动添加的。因此用字符指针指向一个字符串输出时，结果同用字符数组输出结果是一样的。

语句：

```
char *c="c program";
```

与语句：

```
char *c;  
c="c program";
```

是等价的。都是为字符串 "c program" 在内存中开辟连续存储空间，将首地址赋值给指针变量 *c*，并在最后的存储单元中添加字符串结束标志 '\0'。

### 6.13.2 字符串指针作为函数的参数

前面讲过，数组名作为函数的参数实质上是指针常量作为函数的参数，它是指针作为函数参数的一个特例。字符串指针作为函数的参数的实质是指针变量作为函数的参数，它同样是指针作为函数参数的一个特例。看下面这个例子。



**例程 6-20** 编写一个函数，要求把一个字符串的内容复制到另一个字符串中。

```
#include <stdio.h>
copy(char *s,char *q);
int main(void)
{
    char *str,strcp[20],*p;
    str="Hello World!";
    p=strcp;
    copy(str,p);
    printf("%s",strcp);
    getch();
    return 1;
}
copy(char *s,char *q)
{
    int i=0;
    while(*(s+i))
    {
        *q=*(s+i);
        i++;
        q++;
    }
    *q='\0';
}
```

本例程用 copy 函数实现字符串的复制。函数 copy 有两个参数 s 和 q，其功能是将 s 指向的字符串复制到 q 指向的存储空间（字符数组）中。本例程中调用函数 copy 时传递的两个参数是 str 和 p。str 为指向字符串"Hello World!"的指针，p 指向字符数组 strcp 的首单元，即 p 等于 strcp。函数 copy 将 str 指向的字符串"Hello World!"复制到 p 指向的字符数组 strcp 中。

在函数 copy 内部通过循环复制字符串。形参 s 指向原字符串 str，形参 q 指向字符数组 strcp 的首单元。\*(s+i)为原字符串 str 中第 i 个元素，每次循环后 i 自动加 1，s+i 指向下一个元素；每次循环后 q 自增加 1，从而 q 也不断指向下一个元素。循环直到\*(s+i)的值为'\0'，表明原字符串结束，复制完成。最后通过\*q='\0'给字符数组 strcp 添加结束标志，以便使用"%s"格式符输出该字符串。这里要注意，只有在初始化字符串或者用"%s"格式符输入一个字符串时系统才会自动为字符串添加结束标志。

本例程的运行结果为：

```
Hello World!
```

关于本例程有几点说明：

(1) 本例程因为要演示“字符串指针作为函数的参数”，因此加上

```
p=strcp;
```

这一语句，然后将 p 作为参数。p 是一个指针变量，它指向字符数组 strcp 的首单元，值与 strcp 相同。而 strcp 是字符数组名，是个指针常量。其实

```
copy(str,p);
```

等价于

```
copy(str,strcp);
```

(2) 能否将主函数 main 中定义的



```
char *strcpy[20]
```

改为

```
char *strcpy;
```

这样，通过函数 copy 直接向 strcpy 指向的内存空间复制字符串呢？  
用指向字符串的指针初始化一个字符串是可以的，例如：

```
char *str;  
str="Hello World!";
```

但并不提倡应用这种方法非初始化地输入一个字符串，例如：

```
char *a;  
scanf("%s",a); /*直接向 a 所指的内存空间赋值*/
```

或是将它作为函数的参数，例如：

```
char *str,*strcpy; /*定义 str 和 strcpy 为指向字符串的指针*/  
str="Hello World! ";  
copy(str,strcpy); /*将 strcpy 作为参数*/
```

因为 strcpy 中的内容是不确定的，我们并不知道指针变量 strcpy 指向哪里。它不像数组名那样明确的指向系统为该数组开辟的内存空间的首单元，这种定义的指针变量指向的内存空间的位置是不确定的。如果指针变量 strcpy 指向的是内存中的代码段，如图 6-24 所示，那么再向 strcpy 指向的内存空间存储一段数据就会破坏程序，影响系统的正常运行。所以，在用字符串指针作为函数的参数时，字符串指针的指向一定要明确，即字符串指针变量中的内容要确定。

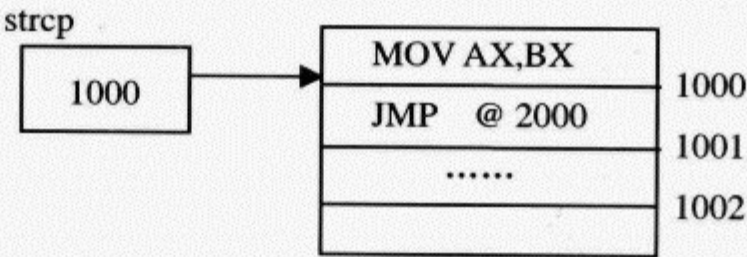


图 6-24 strcpy 指向的是内存中的代码段

6.14 指针与函数

6.14.1 指向函数的指针

一个函数在内存中要占用一段连续的存储空间，函数名就是所占内存空间的首地址。这个首地址也叫做函数的入口地址或函数的指针。在 C 语言中，可以将函数的首地址赋予一个指针变量，使得该指针变量指向此函数，这样就可以通过该指针变量调用这个函数。  
函数指针变量定义的一般形式为：

```
类型说明符 (*指针变量名)();
```

这里的类型说明符是指函数的返回值类型。  
例如：



```
int (*p)();
```

就表示定义了一个指针型变量 `p`，该变量可以存放一个返回值为整型的函数的首地址。这里要注意，`*p` 外面的括号 “`()`” 不要丢掉。

下面通过一个例子来理解指向函数的指针。

**例程 6-21 指向函数的指针演示。**

```
#include <stdio.h>
int main(){
    int max(int a,int b);
    int (*pmax)();
    int x,y,z;
    /*将函数名(函数的首地址)赋值给指针变量 pmax*/
    pmax=max;
    printf("input two numbers:\n");
    scanf("%d%d",&x,&y);
    /*通过指向函数的指针调用函数*/
    z=(*pmax)(x,y);
    printf("max=%d",z);
    getch();
    return 1;
}
int max(int a,int b){
    if(a>b)return a;
    else return b;
}
```

本例程中定义了一个指向函数的指针变量：

```
int (*pmax)();
```

`pmax` 是一个指向函数的指针变量，且该函数的返回值为整型。然后将函数名（函数的首地址）`max` 赋值给指针变量 `pmax`，因此指针变量 `pmax` 指向了函数 `max`，如图 6-25 所示。

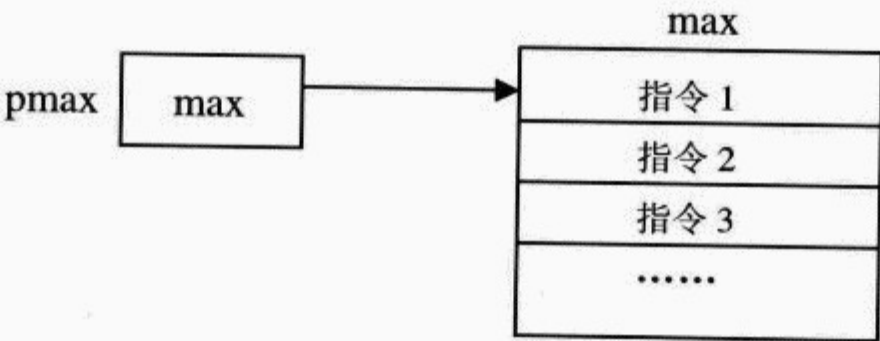


图 6-25 指向函数的指针

通过指向函数的指针变量调用函数时，用到了语句：

```
z=(*pmax)(x,y);
```

它的作用等价于：

```
z=max(x,y);
```

相当于直接调用函数 `max`。其实可以把`(*pmax)`看作函数名 `max`，在函数调用时用`(*pmax)`替代 `max` 即可。但是这里一定要注意，`pmax` 是指向一个“返回值为整型的函数”的指针变量，`(*pmax)(x,y)`的返回值是整型，因此 `z` 一定是整型变量。

本例程的运行结果为：

```
input two numbers:
2 5
max=5
```

### 6.14.2 返回指针的函数

函数不仅可以返回一个整型值、浮点型值、字符型值，同样可以返回一个指针型数据，也就是返回一个地址。

定义指针型函数的一般形式为：

```
类型说明符 *函数名(形参表)
{
    ..... /*函数体*/
}
```

其中函数名之前加了“\*”号表明这是一个指针型函数，返回值是一个指针（地址）。类型说明符表示了返回的指针值所指向的数据类型。

例如：

```
int *func(int a)
{ /*函数体*/
}
```

表示这是一个指针型函数，函数名为 func，函数返回一个指向整数的指针。

下面通过一个例子来理解返回值为指针的函数。

#### 例程 6-22 返回值为指针的函数演示。

```
#include "stdio.h"
int main()
{
    int *Max_Value(int *p,int n);
    int a[10],*max,i;
    printf("Please enter ten integer;\n");
    for(i=0;i<10;i++){
        scanf("%d",&a[i]);
    }
    max=Max_Value(a,10);
    printf("The maxmum is:%d\n",*max);
    getch();
    return;
}
int *Max_Value(int *p,int n)
{
    int i,*max;
    max=p;
    for(i=0;i<n;i++)
    {
        if(*(p+i)>*max)
            max=p+i;
    }
    return max;
}
```

本例程实现从一个整数数组中找出最大的元素。函数 Max\_Value 的功能是找出数组中最大的元素，并返回一个指向该元素的指针。



首先在主函数 `main` 中定义一个长度为 10 的整型数组 `a[10]`，并向该数组输入数据。然后调用函数 `Max_Value` 找出数组中最大的元素。函数 `Max_Value` 有两个实参，一个是数组 `a` 的首地址，也就是数组名；另一个是数组的长度 10。通过指向数组的指针在函数 `Max_Value` 中找出数组 `a` 中的最大值，并将地址赋值给 `max`，最后返回 `max`。

这里应当明确，返回的是数组 `a` 中最大值元素的指针，而非最大值本身。`max` 是主函数中定义的指向整数的指针变量，因此可以将函数的返回值赋值给 `max`。`*max` 为 `max` 指向的数组元素值，也就是数组 `a` 中的最大值。

本例程的运行结果为：

```
Please enter ten integer;
4 7 12 3 8 9 20 12 0 6
The maxmum is:20
```

## 6.15 指针数组和指向指针的指针

### 6.15.1 指针数组

数组中也可以包含指针，即数组的元素都是指针类型的数据，这种数组称为指针数组。指针数组中的每个元素都是一个指针，而且这些指针必须指向相同类型的数据。

一维指针数组的定义形式为：

```
类型说明符 *数组名[数组长度]
```

“类型说明符”为指针所指向的数据的类型，“数组长度”为该数组中可以存放的指针个数。

例如：

```
int *p[10]
```

表示 `p` 是一个指针数组，该数组有 10 个数组元素，每个元素值都是一个指针，指向整型变量。

指针数组中存放的是指针类型的数据元素，也就是地址。因此，`p[i]` 就为指针数组 `p` 中第 `i` 个元素，它是一个指针，`*p[i]` 为指针数组 `p` 中第 `i` 个元素指向的变量内容。

指针数组常用来构造字符串数组。在 C 语言中，字符串可以用指向字符串第一个字符的指针表示，所以字符串数组实际上是指向字符串第一个字符的指针所组成的数组，如图 6-26 所示。

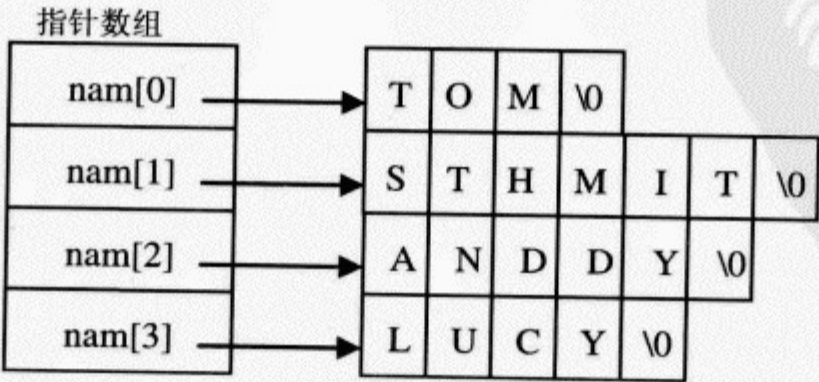


图 6-26 指针数组 `nam` 及指向的字符串

指针数组 `nam` 中，每个元素都是一个指针，指向一个字符串。因此可以把指针数组 `nam` 看作一个字符串数组，每个元素都是一个字符串，但其本质是指针数组。

采用指针数组的好处是节省存储空间且灵活性强。像图 6-26 那样组织学生的姓名是很理想的。数组 `nam` 中存放了学生姓名字符串的指针，方便对学生姓名的管理。如果想将学生的姓名组织成数组，方便管理，但又不想使用指针数组，这时可以用二维数组来存储学生的姓名，如图 6-27 所示。

T	O	M	\0			
S	T	H	M	I	T	\0
A	N	D	D	Y	\0	
L	U	C	Y	\0		

图 6-27 用二维数组存储学生姓名

很显然，用二维数组存储学生姓名字符串时，二维数组的列数必须按最长字符串（`STHMIT`）来定义，这样浪费了许多内存空间。指针数组是一维数组，这样较二维数组更加方便控制。

下面通过例子来理解指针数组。

**例程 6-23** 用指针数组构造字符串数组。

```
#include <stdio.h>
int main(void)
{
    char *Month[]={
        "Illegal Month",
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December",
    };

    int i;
    printf("Please enter a number for Month\n");
    scanf("%d",&i);
    printf("The Month is:");
    printf("%s\n",Month[i]);
    getchar();
    return 0;
}
```

本例程的功能是输入一个月份数，输出该月份的英文表示。



在主函数 main 中定义了指针数组 Month，它包含 13 个元素，分别指向 12 个月份的英文表示和一个 "Illegal Month" 的非法提示。这里用到了对指针数组的初始化，系统会在每个字符串的结尾自动添加字符串结束标志 '\0'。Month[i] 表示指针数组 Month 的第 i 个元素，也就是指向“第 i 月英文表示的字符串”的指针。当 i=0 时，Month[0] 为指向字符串 "Illegal Month" 的指针，表示不存在 0 月份。

在输出字符串时，用到语句：

```
printf("%s\n", Month[i]);
```

Month[i] 为指向字符串的指针，应用 “%s” 格式符输出字符串，不用考虑字符串的长度，系统会以字符串结束标志 '\0' 作为字符串的结尾。

本例程的运行结果如下。

若输入 0：

```
Please enter a number for Month
0
The Month is:Illegal Month
```

若输入 6：

```
Please enter a number for Month
6
The Month is:June
```

其实，指针数组中不仅可以存放指向字符串的指针，还可以存放指向整型变量的指针、指向浮点型变量的指针、指向一般数组的指针、指向结构体的指针甚至还可以存放指向文件的指针。这里只不过是以前指向字符串的指针为例，对指针数组做一个简要的介绍。在一些大型系统的开发中（例如操作系统、数据库管理系统等），指针数组常作为文件的索引，可见指针数组的应用十分广泛，功能也很强大。

### 6.15.2 指向指针的指针

如果一个指针变量存放的是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。直接通过变量名访问变量叫做直接寻址；通过变量的指针访问变量叫做间接寻址；通过指向指针的指针变量来访问变量则构成二级间址，如图 6-28 所示。

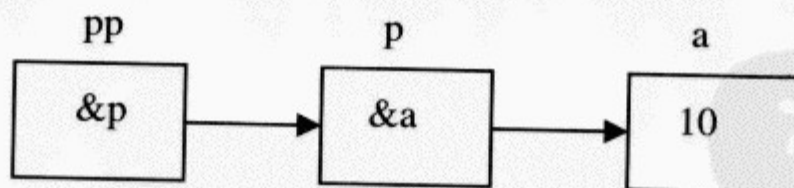


图 6-28 指向指针的指针

a 为一个整型变量，里面存放一个整数 10，p 为一个指针型变量，里面存放变量 a 的地址，即 p 指向变量 a，pp 是一个指向指针的指针变量，里面存放指针变量 p 的地址，即 pp 指向指针变量 p。

指向指针的指针变量定义如下：

```
int **p;
```

它表示 `p` 是一个指向指针的指针变量，`p` 中可以存放一个指针变量的地址，该指针变量中可以存放一个整型变量的地址。

下面通过一个例子来理解指向指针的指针变量。

**例程 6-24 指向指针的指针变量演示。**

```
#include <stdio.h>
int main(void)
{
    char *Month[]={
        "Illegal Month",
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December",
    };

    char **p;
    int i;
    p=Month;
    printf("Please enter a number for Month\n");
    scanf("%d",&i);
    printf("The Month is:");
    printf("%s\n",*(p+i));
    getchar();
    return 0;
}
```

本例程依然沿用例程 6-24，只是采用了指向指针的指针读取字符串。

前面讲过，数组名是数组首单元的指针，因此指针数组的数组名就是指针数组首单元的指针，而指针数组首单元中存放的也是一个指针，因此指针数组的数组名就是一个指针变量的指针，如图 6-29 所示。

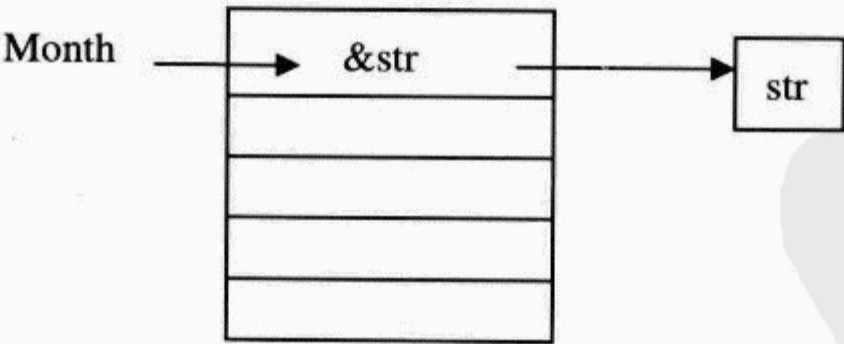


图 6-29 指针数组的数组名

本例程中定义了一个指向字符指针的指针变量：

```
char **p;
```



然后把指针的指针（指针变量的地址）Month 赋值给 p，从而 p 也指向了字符数组的首单元，如图 6-30 所示。

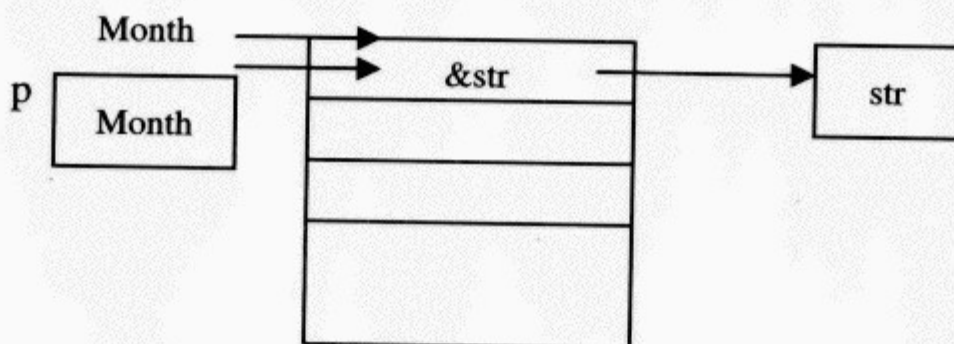


图 6-30 p 指向字符数组的首单元

这里要注意，Month 是指向指针的常量，p 则是指向指针的变量，Month 通过 p 来找到指针数组中的元素。例如：\*(p+2)就表示 Month[2]的内容，它也是一个指针，指向字符串 "February"。因此在输出字符串时，用到语句：

```
printf("%s\n", *(p+i));
```

其中\*(p+i)等价于 Month[i]，它表示指针(p+i)指向的内容，该内容也是一个指针，指向第 i 月的英文单词。

本例程的运行结果为：

```
Please enter a number for Month
5
The Month is:May
```

在实际的编程中很少用到指向指针的指针，因为间址的级数越多，编程出错的几率越大，越容易造成混乱。除非是特别的需要，一般不建议在编程中过多使用指向指针的指针。

## 6.16 void 型指针

ANSI 新标准增加了一种 void 指针类型，可以定义一个指针变量，但不指定它指向哪一种类型数据。

ANSI C 规定，调用动态存储分配函数（例如 malloc）时，返回一个 void 型指针，它可以指向一个抽象的类型数据。将动态存储分配函数的返回值赋值给另一个指针变量时，要进行强制类型转换。

例如：

```
struct stud *p;
p=( struct stud *)malloc(sizeof(struct stud)*10);
```

函数 malloc 返回一个 void 类型指针时，必须将它强制转换为 struct stud \*类型，即指向结构体 struct stud 类型的指针后，才能将它赋值给指针变量 p。

有关结构体和动态存储分配函数的相关知识将在下一章中介绍。



## 6.17 本章小结与要点回顾

本章主要介绍了数组和指针的相关知识。在C语言中，数组和指针是两个极为重要的知识点。数组作为一种构造的数据类型，在数据的组织存储方面有着广泛的应用；指针是C语言的最主要的风格之一，灵活地应用指针可以编写出许多功能强大的程序。数组与指针之间有着非常紧密的联系，因此熟练地掌握数组、指针的相关知识是学好C语言的关键。下面将本章的重点知识归纳如下。

### 1. 一维数组的定义

在C语言中，一维数组定义的一般形式为：

类型说明符 数组名 [常量表达式]；

### 2. 一维数组的元素

在C语言中，数组元素可表示为：

数组名[下标]

其中下标只能为整型常量或整型表达式。如果是小数，C编译将自动取整。

### 3. 一维数组的初始化

对数组元素进行初始化的一般形式为：

类型说明符 数组名 [常量表达式] = {值, 值……值};

其中“{ }”中的各数据值为元素的初值，它们应是同一类型，各值之间用逗号间隔。

### 4. 二维数组的定义

二维数组定义的一般形式是：

类型说明符 数组名 [常量表达式 1] [常量表达式 2]

“类型说明符”用来说明数组中元素的数据类型。“数组名”是唯一标识该二维数组的名字。第一个方括号“[]”中的常量表达式1表示第一维下标的长度，第二个方括号“[]”中常量表达式2表示第二维下标的长度。

### 5. 二维数组的元素

二维数组的元素也称为双下标变量，其表示的形式为：

数组名[下标][下标]

其中下标应为整型常量或整型表达式。

### 6. 二维数组的初始化

有两种方法对二维数组进行初始化。

(1) 按行分段赋值。例如：



```
int a[2][3]={{1,2,3},{4,5,6}};
```

把每一行的元素写在一个花括号“{}”中。

(2) 按行连续赋值。例如：

```
int a[2][3]={1,2,3,4,5,6};
```

将二维数组看成一个连续的空间，其效果与第一种方法相同。

## 7. 指针的概念

指针就是内存的地址。

## 8. 定义指针变量

其一般形式为：

类型说明符 \*指针变量名；

其中，“类型说明符”表示指针变量所指向变量的数据类型，也就是该指针变量的基类型。“\*”表示这是一个指针变量，有别于其他的变量。

## 9. 指针变量的引用

指针运算符“\*”在定义指针变量时做指针类型说明，表明定义的变量是指针型变量，用于间接访问指针变量所指向的内存单元。

## 10. 指针变量作为函数的参数

函数的参数不仅可以是基本类型的变量（整型变量、字符型变量等），还可以是指针类型的变量。其作用是将一个变量的地址传递给被调函数。这样就可以在被调函数中通过指针（地址）操纵主调函数中的变量。

## 11. 指向数组元素的指针

(1) 定义指向数组元素的指针变量的方法与定义指向变量的指针变量的方法相同。

(2) 数组名代表数组的首地址。

## 12. 用指针引用数组元素

引用数组的元素有两种方法：一是通过数组的下标引用；二是通过指向数组的指针引用。

## 13. 数组名作为函数的参数

数组名的本质是数组的首地址，因此可以将数组名作为函数的参数传递给被调函数，这样就可以在被调函数中对主调函数中定义的数组进行访问和操作。

## 14. 二维数组的指针

二维数组指针变量说明的一般形式为：

类型说明符 (\*指针变量名)[长度]

其中，“类型说明符”为所指数组的数据类型。“\*”表示其后的变量是指针类型。“长



度”表示二维数组分解为多个一维数组时的一维数组长度，也就是二维数组的列数。

### 15. 字符数组

(1) 字符数组的定义：与数值数组定义相同。

(2) 字符数组的初始化：与整型数组，浮点型数组等一样，允许在定义时作初始化赋值。

(3) 字符串的结束标志：C语言中以'\0'作为字符串的结束标志。

(4) 字符数组的输入输出：逐个字符的输入输出或利用格式符“%s”一次性的输入输出。

### 16. 字符串指针

(1) 用指针指向字符串：可以用字符指针指向一个字符串代替定义字符数组。

(2) 字符串指针作为函数的参数：字符串指针作为函数参数的实质是指针变量作为函数的参数。

### 17. 指向函数的指针

函数指针变量定义的一般形式为：

```
类型说明符 (*指针变量名)();
```

这里的“类型说明符”是指函数的返回值类型。

### 18. 返回指针的函数

函数可以返回一个整型值、浮点型值、字符型值，同样可以返回一个指针型数据，也就是返回一个地址。

### 19. 指针数组

(1) 数组中也可以包含指针，即数组的元素都是指针类型的数据，这种数组称为指针数组。

(2) 一维指针数组的定义形式为：

```
类型说明符 *数组名[数组长度]
```

“类型说明符”为指针所指向的数据的类型，“数组长度”为该数组中可以存放的指针的个数。

### 20. 指向指针的指针

(1) 如果一个指针变量存放的是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。

(2) 指向指针的指针变量定义如下：

```
int **p;
```

它表示 p 是一个指向指针的指针变量，p 中可以存放一个指针变量的地址，该指针变



量中可以存放一个整型变量的地址。

## 21. void 型指针

void 指针类型是 ANSI 新标准增加的类型。void 类型指针并不指定它是指向哪一种类型数据，而是根据需要强制转换为需要的数据类型。

在前面的章节中介绍了 C 语言中的很多数据类型，包括基本数据类型（整型、字符型、浮点型、枚举类型），构造数据类型（数组型），指针类型和空类型。但只有这些数据类型是不够的，因为它们难以胜任复杂的程序设计和软件开发。

本章将介绍另外两种构造数据类型：结构体和联合。它们是在上述介绍的数据类型的基础上构造的更为复杂的数据类型，因此功能更为强大。

## 7.1 结构体概述

在实际的工作中，有些数据的组织形式很复杂，它们很难用前面几章介绍的数据类型直接表示。例如学生信息管理系统中经常要处理的“学生信息记录”就是一类复杂的数据。因为在每个学生记录中，都要包括学生的姓名、性别、年龄、学号、成绩等信息，而姓名应为字符型；学号可为整型或字符型；年龄应为整型；性别应为字符型；成绩可为整型或浮点型。显然不能只用一种数据类型定义这些数据，也不能用一个数组来存放数据，因为数组中各元素的类型和长度都必须一致。为了解决这个问题，C 语言中给出了另外一种构造数据类型——“结构（structure）”，也称为“结构体”。

结构体是一种构造类型。通过对数组的学习不难理解它应该像数组那样由若干元素组成。但在这里，这些元素被称为“成员”，每一个成员既可以是一个基本数据类型又可以是一个构造类型。

作为一种构造的数据类型，结构体并不是系统定义的，而是由用户自己定义的。因此，使用结构体之前必须先定义。

定义的一般形式为：

```
struct 结构名  
{成员表列};
```

成员表列由若干个成员组成，每个成员都是该结构的一个组成部分。对每个成员也必须作类型说明，其形式为：

类型说明符 成员名；

举一个具体的例子：

```
struct stu  
{  
    int num;
```



```
char name[20];  
char sex;  
float score;  
};
```

上面的这段代码表示定义了一个结构体 `stu`，该结构体中有 4 个成员，包括 1 个整型变量 `num`，一个字符串数组 `name`，一个字符型变量 `sex`，一个浮点型变量 `score`。这样就可以用这个结构体表示一个学生的记录了。

必须明确的一点是，上面定义的这个结构体只相当于一个类型，而并非一个实体变量。它同整型（`int`）、字符型（`char`）等一样，只是一个类型而已，不过它是用户自己定义的构造类型，而非系统定义的类型。

## 7.2 结构体变量的定义

前面讲到，用 `struct` 定义的结构体只是一个类型，系统根据定义的类型为之分配空间。在没有为结构体定义变量之前，结构体只作为一个用户自定义的类型而抽象地存在，并没有具体实例化。因此，要使用结构体组织数据，就必须定义结构体变量。

本节将介绍三种方法定义结构体类型变量。

### 7.2.1 先定义类型后定义变量

在定义结构体变量时，可以先定义结构体类型，再定义结构体变量。例如：

```
struct stu  
{  
    int num;  
    char name[20];  
    char sex;  
    float score;  
};  
struct stu boy1, boy2;
```

先定义了一个结构体类型 `struct stu`，再定义两个结构体变量 `boy1` 和 `boy2`。这里要注意定义结构体变量的格式：

结构体类型名 结构体变量名 1, 结构体变量名 2...;

对应上面的定义，`struct stu` 为结构体类型名，`boy1`、`boy2` 为结构体变量名。

定义了结构体变量，系统就要为该变量分配内存空间，这一点与定义其他类型变量是一样的。系统为结构体变量分配空间的大小与用户定义的结构体类型有关。像上面这个例子，因为 `stu` 类型中包含 1 个整型变量成员 `num`，一个字符串数组成员 `name`，一个字符型变量成员 `sex`，一个浮点型变量成员 `score`，因此系统为 `stu` 类型的变量 `boy1`，`boy2` 各分配（ $2+20+1+4=27$ ）字节大小的存储空间。

其实每个结构体类型的变量都是用户定义的结构体类型的一个具体的实例、一个对象。例如：变量 `boy1` 中就包含它自己的 1 个整型变量成员 `num`，一个字符串数组成员 `name`，一个字符型变量成员 `sex`，一个浮点型变量成员 `score`；变量 `boy2` 中也包含相同的上述成员，



但 boy1 与 boy2 之间没有任何联系。结构体类型只是一个抽象的类型，它同整型、浮点型是一个逻辑层面上的概念。理解了这一点，就不难理解结构体类型和结构体变量，以及为什么要定义结构体变量了。

为了方便起见，也可以这样定义结构体类型和结构体变量。

```
#define STU struct stu
STU
{
    int num;
    char name[20];
    char sex;
    float score;
};
STU boy1, boy2;
```

这里用到了宏定义，将字符串 struct stu 定义为字符常量 STU，这样在程序中就可以用 STU 代替 struct stu 了。

### 7.2.2 在定义结构体类型的同时定义变量

在定义结构体类型的同时也可以定义结构体变量，它的一般形式为：

```
struct 结构名
{
    成员表列
}变量名表列;
```

例如：

```
struct stu
{
    int num;
    char name[20];
    char sex;
    float score;
}boy1, boy2;
```

这段代码在定义结构体类型 struct stu 的同时定义了结构体类型变量 boy1 和 boy2。

采用这种方法定义结构体变量多用作全局变量，因为一般情况下结构体类型的定义在函数之外（也可以定义在函数内部）。因此，在定义全局变量的结构体时可以采用这个方法。

### 7.2.3 直接定义结构体变量

还可以直接定义结构体变量，而不需要给出结构体名，它的一般形式为：

```
struct
{
    成员表列
}变量名表列;
```

例如：

```
struct
{
    int num;
```



```
    char name[20];  
    char sex;  
    float score;  
}boy1,boy2;
```

直接定义结构体变量 boy1 和 boy2, 而不需要出现结构体名。

以上三种定义结构体变量的方法, 只是在不同的位置上定义变量, 变量的本质是相同的。每个变量都包含 4 个成员, 变量在内存中所占空间大小都相等 (27 字节)。定义好的结构体变量才是真正的学生信息记录的载体, 才能对它进行赋值运算。

#### 7.2.4 关于结构体类型的几点说明

下面对结构体类型进行几点说明:

(1) 结构体中的成员也可以是一个结构体变量, 即构成了嵌套的结构。

例如:

```
struct date  
{  
    int month;  
    int day;  
    int year;  
};  
struct stu {  
    int num;  
    char name[20];  
    char sex;  
    struct date birthday;  
    float score;  
}boy1,boy2;
```

结构体类型 struct stu 中包含 struct date 类型的成员变量 birthday, 用来表示学生的生日。已声明的类型 struct date 与其他类型 (例如 int) 一样, 都可以用来定义成员变量的类型。

(2) 成员名可以与程序中的变量名相同。

例如:

```
struct stu  
{  
    int num;  
    char name[20];  
    char sex;  
    float score;  
};  
Int main()  
{  
    struct stu boy1,boy2;  
    int num;  
    ... ..  
}
```

在主函数中定义了整型变量 num, 它可以与结构 struct stu 中的成员变量 num 重名, 并不发生冲突。

## 7.3 结构体变量的引用

使用结构体变量的关键是使用它的成员变量，结构体其本质是不同类型的数据成员的集合。因此对结构体变量的引用，归根到底是对结构体变量中的数据成员的引用。

### 7.3.1 结构体成员的引用

定义了结构体变量之后，可对其中的成员赋值。引用结构体变量中成员的方式为：

变量名.成员名

例如：

boy1.num

表示结构体变量 boy1 中的 num 成员。其中“.”是成员分量运算符，它的优先级在所有运算符中是最高的，因此赋值语句

boy1.num=1000;

不用写成

(boy1.num)=1000;

下面通过一个实例来理解结构体变量的引用。

#### 例程 7-1 结构体变量的引用。

```
#include <stdio.h>
struct stu
{
    int num;
    char name[10];
    char sex;
    float score;
};
int main(void)
{
    struct stu boy;
    /*引用结构体变量为之赋值*/
    boy.num=1000;
    strcpy(boy.name, "Jam");
    boy.sex='M';
    boy.score=85.5;
    /*打印出该学生信息*/
    printf("\nThe information of the boy is\n");
    printf("num: %d\n", boy.num);
    printf("name: %s\n", boy.name);
    printf("sex: %c\n", boy.sex);
    printf("score: %f\n", boy.score);
    getchar();
    return 0;
}
```

本例程中定义了一个结构体变量 boy，然后为它的成员赋值。其中 strcpy(boy.name, "Jam") 是调用字符串处理函数，将字符串 "Jam" 复制到以 boy.name 为首地址的字符串数组中。输



出学生姓名信息时用“%s”格式输出符。这样一个完整的学生信息记录就创建成了。

boy 是一个结构体变量，它是一个数据实体，里面包含着 num 等 4 类信息。通过“.”运算符可以直接引用（添加、修改）boy 中的各个成员。

本例程的运行结果为：

```
The information of the boy is
num: 1000
name:Jam
sex: M
score:85.500000
```

结构体变量中的成员也可以是结构体。对结构体成员的成员引用时，要用到若干成员运算符，这样一级一级地找到最低一层成员。例如：

```
struct date
{
    int month;
    int day;
    int year;
};
struct stu {
    int num;
    char name[20];
    char sex;
    struct date birthday;
    float score;
}boy1,boy2;
```

其中，boy1 和 boy2 就是上述的结构体变量。如果要修改 boy1 中的 birthday 成员中的 month 值，就要用到若干成员运算符：

```
boy1.birthday.month=3;
```

这里将 boy1 中的 birthday 成员中的 month 成员赋值为 3。

当然，对结构体变量成员赋值不一定像上面那样在程序中给出，可以通过 scanf 语句从终端输入数据。它的方法同一般变量的数据输入是一样的，要引用数据成员的地址。看下面这个例子。

### 例程 7-2 输入一个学生记录并打印出来。

```
#include <stdio.h>
struct stu
{
    int num;
    char name[10];
    char sex;
    float score;
};
int main(void)
{
    struct stu boy;
    /*引用结构体变量为之赋值*/
    printf("Input num\n");
    scanf("%d",&boy.num);
    printf("Input name\n");
    scanf("%s",boy.name);
```



```
    getchar();
    printf("Input sex\n");
    scanf("%c",&boy.sex);
    printf("Input score\n");
    scanf("%f",&boy.score);
    /*打印出该学生信息*/
    printf("\nThe information of the boy is\n");
    printf("num: %d\n",boy.num);
    printf("name:%s\n",boy.name);
    printf("sex: %c\n",boy.sex);
    printf("score:%f\n",boy.score);
    getchar();
    return 0;
}
```

本例程中，通过 `scanf` 语句对结构体变量进行赋值。需要指出的是，`scanf("%s",boy.name)` 语句中的 `boy.name` 本身就是一个地址，它表示结构体成员 `name` 数组的首地址，因此不用加取地址符“&”。本句后面的语句 `getchar()` 的作用是接收来自终端的一个字符（回车符），以区别下面要输入的字符信息。本例程的运行结果为：

```
Input num
1000
Input name
Jam
Input sex
M
Input score
85.5

The information of the boy is
num: 1000
name:Jam
sex: M
score:85.500000
```

### 7.3.2 结构体变量的初始化

定义结构体变量时也可以对其成员进行初始化。看下面这个例子。

#### 例程 7-3 结构体变量的初始化。

```
#include <stdio.h>
int main()
{
    struct stu    /*定义结构*/
    {
        int num;
        char name[10];
        char sex;
        float score;
    }boy={1000,"Jam",'M',85.5};
    printf("\nThe information of the boy is\n");
    printf("num: %d\n",boy.num);
    printf("name:%s\n",boy.name);
    printf("sex: %c\n",boy.sex);
    printf("score:%f\n",boy.score);
    getchar();
    return 0;
}
```



与前面的例子不同，本例程在主函数中定义结构体类型和结构体变量 boy，并为变量 boy 赋初值。然后用 printf 语句输出结果。结构体变量初始化的方法与数组初始化的方法类似，要用到一个大括号“{}”，大括号外面的分号“;”不能省略。本例程的运行结果为：

```
The information of the boy is
num: 1000
name:Jam
sex: M
score:85.500000
```

## 7.4 结构体数组

同一般类型的变量一样，结构体类型的变量也可以组织成数组，称之为结构体数组。不同之处在于以前介绍的数组中，数组元素都是一个基本类型的变量，而结构体数组中的每个数组元素都是用户自定义的结构体类型变量。

### 7.4.1 结构体数组的定义

结构体数组的定义与一般类型的数组定义类似，例如：

```
struct stu student[10];
```

表示定义了一个 struct stu 结构体类型的结构体数组 student，student 中共有 10 个元素，每个元素都是一个 struct stu 类型的结构体变量。

与定义结构体变量一样，也可以直接定义结构体数组：

```
struct stu
{
    int num;
    char name[10];
    char sex;
    float score;
}student[10];
```

或者：

```
struct
{
    int num;
    char name[20];
    char sex;
    float score;
} student[10];
```

定义好的结构体数组 student 在内存中的形式如图 7-1 所示。

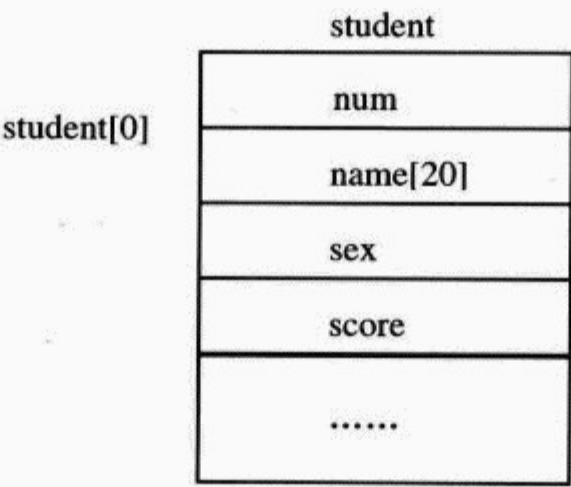


图 7-1 结构体数组 student 在内存中的形式

7.4.2 结构体数组的初始化

结构体数组也可以进行初始化，初始化的方式类似于二维数组的初始化。例如：

```
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}boy[5]={
    {101,"Li ping","M",45},
    {102,"Zhang ping","M",62.5},
    {103,"He fang","F",92.5},
    {104,"Cheng ling","F",87},
    {105,"Wang ming","M",58};
}
```

即是在定义结构体数组的同时对结构体数组中的元素进行初始化。当对全部元素做初始化赋值时，也可不给出数组长度。例如：

```
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}boy[]={
    {101,"Li ping","M",45},
    {102,"Zhang ping","M",62.5},
    {103,"He fang","F",92.5},
    {104,"Cheng ling","F",87},
    {105,"Wang ming","M",58};
}
```

系统会自动开辟 5 个 stu 类型大小的内存空间来存放初始化的数据。

7.4.3 结构体数组举例

**例程 7-4** 对学生信息记录按学号从小到大的顺序排序。

分析：对一般类型的数组排序，可以选用“冒泡排序”的算法，也可以采用“选择排



序”的算法，这些排序的方法对于结构体数组来说同样适用。只不过这里比较的元素是每个结构体变量中成员的学号。为了输入输出的方便，这里只给每个学生记录设定两个成员，即学号和姓名。下面给出程序代码。

```
#include <stdio.h>
#define MAX 6
struct stu{
    int num;
    char name[5];
};
int main(void)
{
    int i,j;
    struct stu sort[MAX],min,t;
    printf("Please input six student records:\n");
    for(i=0;i<MAX;i++)
    {
        printf("Number:");
        scanf("%d",&sort[i].num);
        printf("Name:");
        scanf("%s",sort[i].name);
    }
    for(i=0;i<MAX-1;i++)
    {
        min=sort[i];
        for(j=i+1;j<MAX;j++)
            if(sort[j].num<min.num)
            {
                t=min;
                min=sort[j];
                sort[j]=t;    /*swap*/
            }
        sort[i]=min;
    }
    printf("The result of sort is:\n");
    for(i=0;i<MAX;i++)
        printf("%d,%s\n",sort[i].num,sort[i].name);
    getchar();
    return 0;
}
```

本例程是对例程 6-2 的改进，将比较的元素由整型变为结构体的成员变量 `num`。首先输入 6 组学生记录，包括学生的学号和姓名，然后程序按照学生的学号从小到大进行排序，最后输出排序后的结果。本例程的运行结果为：

```
Please input six student records:
Number:6
Name:Wu
Number:5
Name:Zhou
Number:4
Name:Li
Number:3
Name:Sun
Number:2
Name:Qian
Number:1
Name:Zhao
The result of sort is:
```



```
1,Zhao
2,Qian
3,Sun
4,Li
5,Zhou
6,Wu
```

需要注意的是，在定义结构体类型时将姓名成员定义为：

```
char num[5];
```

这样一个字符数组，因此就要求每个记录中的姓名一项不得超过 5 字节。那么，可否采用如下的定义形式呢？这样就没有姓名字符数的限制了。

```
struct stu{
    int num;
    char *name;
};
```

答案是否定的。在 6.4.2 节中曾解释过这个原因，因为接下来要通过 scanf 语句向 name 所指向的空间输入字符串，然而用户并不知道 name 指向的空间是哪一段空间以及是否可用。这种定义的指针变量指向内存空间的位置是不确定的，因此并不提倡这样使用字符串指针。而 name[5]的 name 是一个确定的指针常量，可以放心使用。

本例程旨在演示结构体数组的应用。在实际的使用中，结构体数组是一类很有用的数据结构，它可以装载从文件中读入的数据记录作为内存缓冲区使用。例如：将一个文件中的学生记录读入到结构体数组中，再对其按学号的降序重新排列顺序，然后再写回到文件之中，作为学生信息管理系统的一部分。

## 7.5 指向结构体的指针

与其他类型的变量一样，也存在指向结构体变量的指针。本节将介绍指向结构体变量的指针和基于结构体指针而构造出的一种较为复杂的数据结构——链表。

### 7.5.1 指向结构体变量的指针

一般指针也可以指向结构体变量。这个指针就是该结构体变量占据的内存空间的起始地址。如图 7-2 所示。

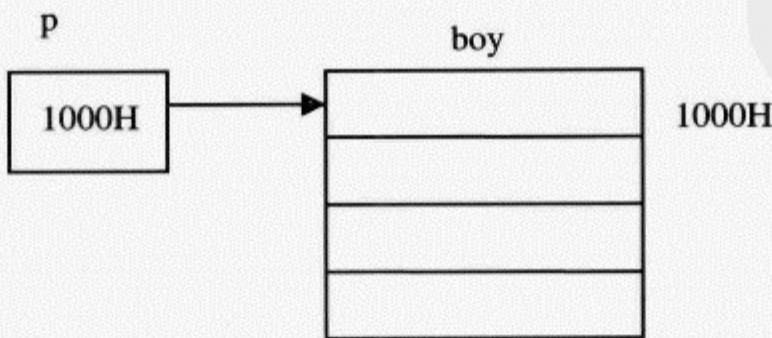


图 7-2 指向结构体变量 boy 的指针



图 7-2 中, boy 是定义好了的一个结构体变量, 它在内存中占据一段内存空间。1000H 为该结构体变量在内存中的起始地址, 也就是该结构体变量的指针。p 为一个指向结构体变量的指针变量, p 中存放的内容是地址 1000H, 也就是说指针变量 p 指向结构体变量 boy。

同样, 指向结构体变量的指针变量也可以指向结构体数组的元素。

定义一个指向结构体类型的指针变量的一般形式为:

```
struct 结构体名 *变量名;
```

例如:

```
struct stu *p;
```

就表示 p 是一个指向结构体类型 struct stu 的指针变量。p 中可以存放一个结构体变量的指针 (地址)。

通过指向结构体变量的指针 p 也可以引用结构体中的成员。有以下两种方法:

```
(*p).成员名;  
p->成员名;
```

这两种方法是等价的, 都表示 p 指向的结构体变量中的成员。但一般建议使用 “->” 的形式引用结构体变量中的成员, 因为这样显得更为直观。

下面通过例子来理解指向结构体变量的指针。

#### 例程 7-5 指向结构体变量的指针演示。

```
#include <stdio.h>  
struct stu  
{  
    int num;  
    char *name;  
    char sex;  
    float score;  
};  
int main(void)  
{  
    struct stu *p, boy={1, "Tom", 'M', 93.5};  
    p=&boy;  
    printf("Number:%d\n", p->num);  
    printf("Name:%s\n", p->name);  
    printf("Sex:%c\n", p->sex);  
    printf("Score:%f\n", p->score);  
    getchar();  
    return 0;  
}
```

本例程中定义了一个结构体类型变量 boy, 并为其赋初值, 然后将变量 boy 的地址赋值给指针变量 p, 即此时 p 指向了结构体变量 boy。输出时通过 “->” 形式引用 boy 中的数据成员。本例程的输出结果为:

```
Number:1  
Name:Tom  
Sex:M  
Score:93.500000
```



在引入指向结构体变量的指针的概念后，再定义结构体变量时可以不指定变量名，直接用一个指针指向它，不过需要用到动态存储分配函数。下面通过实例来理解这一点。

**例程 7-6 动态分配结构体空间。**

```
#include <stdio.h>
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
};
int main(void)
{
    struct stu *p;
    p=(struct stu *)malloc(sizeof(struct stu)*1);
    p->num=1;
    p->name="Tom";
    p->sex='M';
    p->score=93.5;
    printf("Number:%d\n",p->num);
    printf("Name:%s\n",p->name);
    printf("Sex:%c\n",p->sex);
    printf("Score:%f\n",p->score);
    getchar();
    return 0;
}
```

本例程中没有直接定义结构体变量，而只是定义了一个指向结构体类型的指针变量 `p`。但在程序中用到了动态存储分配函数 `malloc`：

```
p=(struct stu *)malloc(sizeof(struct stu)*1);
```

作用是在内存的动态存储区中分配一个长度为 `sizeof(struct stu)*1`，即一个结构体变量大小的存储空间。6.7 节中曾提到，函数 `malloc` 返回一个 `void` 型指针。这里的指针是分配域的起始地址，其类型为 `void`，所以要将它强制转换为指向结构体的指针才能赋值给指针变量 `p`，如图 7-3 所示。

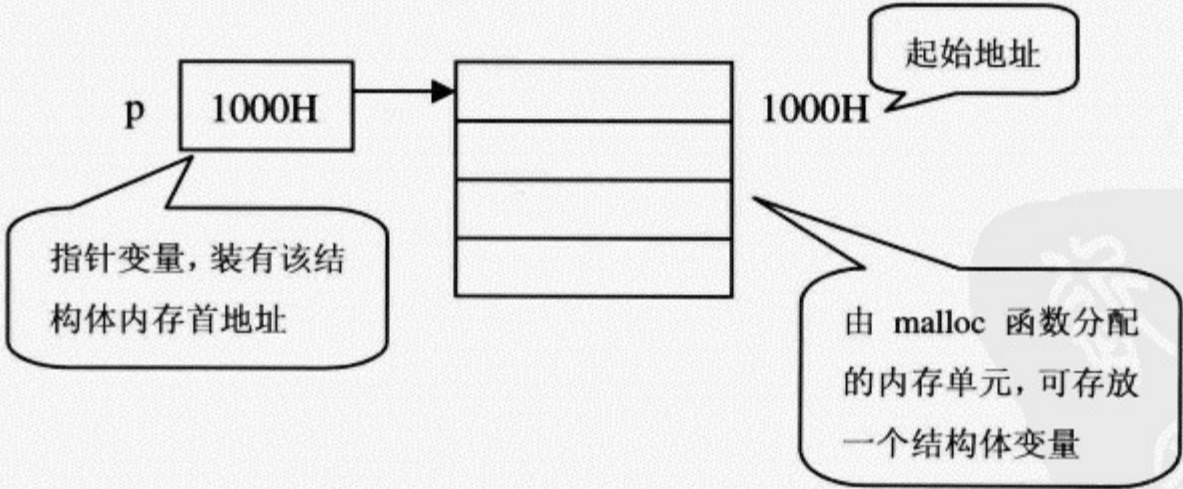


图 7-3 动态分配结构体变量

然后通过指向该结构体变量的指针 `p` 来访问结构体中的成员。注意，这个结构体变量没有变量名，指针 `p` 是该结构体变量中成员的唯一访问渠道。本例程的运行结果为：



```
Number:1
Name:Tom
Sex:M
Score:93.500000
```

这种用动态存储分配函数分配结构体变量的存储空间，并通过指针指向该空间而不去指定变量名的做法是有好处的，这会使得程序设计更加灵活，更节省空间。因为动态存储分配函数 malloc 是在内存的动态存储区中分配空间的，因此它可以随时分配内存单元，不用像定义一般的变量那样必须在程序的开始定义。而且用函数 malloc 申请到的内存空间还可以用函数 free 释放掉，不用像一般的局部变量那样必须在它所在的函数调用完成后才释放掉，从而节省了内存空间。

7.5.2 链表简介

在程序设计中，有一种常见的线性数据存储结构称为“链表”，它是通过在每个结构体变量（或其他构造类型的变量）中设置指向下一个变量的指针域，并存入下一个变量的地址，实现将一组结构体变量（或其他构造类型的变量）连成一条链表。

链表中每个结构体变量称为该链表的一个结点（node），每个结点的单元一般都没有名字，它们是通过动态存储分配函数生成的。

作为一种线性存储结构，链表的使用比数组灵活得多（前面所讲的数组都是在静态存储区中定义的数组）。因为构造一个链表时不一定在程序代码中指定链表的长度（结点的个数），可以在程序的运行过程中动态地生成一条链表。而且，链表使用完后还可以通过函数 free 释放掉整个链表的存储空间。链表的结构如图 7-4 所示。

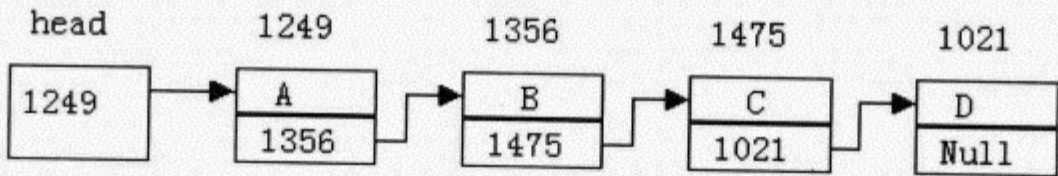


图 7-4 链表的结构

其中第 0 个结点称为整个链表的头结点（head），它一般不存放具体数据，只是存放第一个结点的首地址，因此它只是一个指针变量。最后一个结点的指针域设为空（Null），表示该链表到此结束，没有后续结点。

这里只是简单地介绍一下链表的相关知识，旨在说明指向结构体指针的用处。有关链表的详细知识可参看第 3 部分 16.5 节。

7.6 联合的概念及定义

联合（union）又叫共用体，是 C 语言中定义的一种构造数据类型。它利用覆盖技术，将几种不同类型的变量存放到同一段内存单元中。例如：将一个整型变量，一个浮点型变量，一个字符型变量都放在同一个地址的内存单元中。

覆盖技术就是在同一地址的内存空间中，不同的时间存放不同类型的数据，它不像一



般的变量那样存储定义后不能改变。覆盖技术能充分利用资源,节省存储空间,在最初配置不高的计算机系统中十分有用。

定义联合类型变量的一般形式为:

```
union 联合名  
{成员表列;  
}变量表列;
```

例如:

```
union var  
{  
    int x;  
    char y;  
    float z;  
} a,b;
```

同定义结构体变量一样,也可以如下定义联合类型变量:

```
union var  
{  
    int x;  
    char y;  
    float z;  
}  
union var a,b;
```

或者:

```
union  
{  
    int x;  
    char y;  
    float z;  
} a,b;
```

它们表明 a, b 是两个联合类型的变量。

必须严格区别结构体变量和联合变量。结构体变量包含了结构体类型定义中的所有成员变量。例如:

```
struct var{  
int x;  
char y;  
float z;  
}a;
```

结构体变量 a 中包含了三个成员变量 x, y, z。因此 a 的大小为  $2+1+4=7$  字节。而联合:

```
union var  
{  
    int x;  
    char y;  
    float z;  
} a;
```

变量 a 中每一时刻至多存储一种类型的成员变量,整型 x、字符型 y 或者是浮点型 z。变量 a 的大小应为最长的成员长度,即 z 的长度,为 4 字节。

联合是指几种不同类型的成员变量可以在同一个内存空间中存储,但并不意味着同时



存储这几种不同类型的数据，而是应用覆盖技术在不同的时刻存储不同类型的数据。

## 7.7 联合变量的引用

定义了联合变量就可以引用它的成员，并对其成员变量进行操作。联合变量的数据成员的引用方式同结构体一样，一般形式为：

变量名.成员名

例如：

```
a.x;    /*变量 a 的 x 成员*/  
a.y;    /*变量 a 的 y 成员*/  
a.z;    /*变量 a 的 z 成员*/
```

注意，虽然联合变量中只有一个存储单元用来存放不同类型的变量（整型  $x$ 、字符型  $y$  或者是浮点型  $z$ ），但是对其进行赋值时，不能像对一般变量那样直接赋值，例如：

```
a=1;    /*企图将整数 1 赋值给联合变量 a 中的 x 域*/  
a='c';  /*企图将字符 'c' 赋值给联合变量 a 中的 y 域*/
```

这些都是非法的操作。

下面通过一个实例来理解联合的用法。

### 例程 7-7 联合用法的演示。

```
#include <stdio.h>  
union var  
{  
    int x;  
    char y;  
    float z;  
};  
int main(void)  
{  
    union var a;  
    a.x=1;  
    printf("%d\n",a.x);  
    a.y='c';  
    printf("%c\n",a.y);  
    a.z=1.5;  
    printf("%f\n",a.z);  
    printf("%d",sizeof(union var));  
    getchar();  
    return 0;  
}
```

本例程中定义了一个 `union var` 类型的变量 `a`，然后分别向它的三个数据成员赋值并打印出来，最后显示出该联合变量 `a` 的大小。本例程的运行结果为：

```
1  
c  
1.500000  
4
```



从结果中不难发现，虽然该联合变量有三个不同类型的成员变量，但其大小仅为其中最长的成员长度 4。这说明联合变量的大小并不取决于成员变量的多少，而取决于成员变量中最长成员的长度。

## 7.8 使用联合的注意事项

在使用联合类型的变量时要注意以下几点：

(1) 联合变量中起决定作用的是最后一次赋值。

因为在存入一个新成员后，原成员就会被覆盖掉，从而失去作用，因此只有最后的一次赋值才起决定作用。例如：

```
a.x=1;
a.y='c';
a.z=1.5;
```

最后联合变量 a 中只存有一个浮点数 1.5。因此，在引用联合变量时要特别注意当前联合变量中存放的是哪个成员。

(2) 联合变量的地址与其成员变量的地址是一样的。

联合变量的地址与它的成员变量的地址都是一样的，例如：&a.x, &a.y, &a.z, &a 都是同一个地址。

(3) 不能直接给联合变量赋值

虽然联合变量中只有一个存储单元，但是对其进行赋值时，不能像对一般变量那样直接赋值，要根据赋值数据的类型赋值给不同的成员变量。

(4) 联合变量不能作为函数的参数，函数的返回值也不能是联合类型。

## 7.9 自定义类型

C 语言不仅提供了丰富的数据类型，而且还允许用户自己来定义类型说明符，以方便自己的使用。类型定义符 typedef 可以完成这个功能。

typedef 定义的一般形式为：

```
typedef 原类型名 新类型名
```

其中“原类型名”中含有定义部分，“新类型”名一般用大写表示，以便于区别。

例如整型变量 a, b 其说明如下：

```
int a,b;
```

其中 int 是整型变量的类型说明符。为了增加程序的可读性，可把整型 int 说明符用 typedef 定义为：

```
typedef int INTEGER
```

以后就可用 INTEGER 来代替 int 作整型变量的类型说明了。



例如：

```
INTEGER a,b;
```

等价于：

```
int a,b;
```

使用 `typedef` 来定义数组、指针、结构等类型不仅方便了编程，而且提高了程序的可读性。例如：

```
typedef char NAME[20];
```

表示 `NAME` 是字符数组类型，数组长度为 20。然后可用 `NAME` 说明变量，如：

```
NAME a1,a2;
```

等价于：

```
char a1[20],a2[20];
```

又如：

```
typedef struct stu
{
    int num;
    char *name;
    char sex;
    float score;
} STU;
```

定义 `STU` 表示 `stu` 的结构类型，然后可用 `STU` 来说明结构体变量：

```
STU boy1,boy2;
```

有时也可用宏定义来代替 `typedef` 的功能，但是宏定义是由预处理完成的，而 `typedef` 则是在编译时完成的，后者更为灵活方便。

## 7.10 本章小结与要点回顾

本章介绍了两种构造数据类型：结构体和联合。它们是更为复杂的数据类型，是在基本数据类型的基础上由用户构造而成的。

### 1. 结构体类型的定义

定义一个结构体的一般形式为：

```
struct 结构名
{成员表列};
```

成员表列由若干个成员组成，每个成员都是该结构的一个组成部分。

### 2. 定义结构体类型变量的三种方法

(1) 先定义结构体类型，再定义结构体变量。

```
struct 结构名
```

```
{成员表列};  
struct 结构名 结构体变量名1, 结构体变量名2...;
```

(2) 在定义结构体类型的同时定义结构体变量。

```
struct 结构名  
{  
    成员表列  
}变量名表列;
```

(3) 直接定义结构体变量。

```
struct  
{  
    成员表列  
}变量名表列;
```

### 3. 结构体变量的引用方法

引用结构体变量中成员的方式为：

```
变量名.成员名
```

其中“.”是成员分量运算符，它的优先级在所有运算符中是最高的。

另外，结构体变量中的成员也可以是结构体。对结构体成员的成员进行引用时，要用到若干成员运算符，这样一级一级地找到最低一层成员。

### 4. 结构体数组

结构体类型的变量也可以组织成数组，称之为结构体数组。

定义一个结构体数组的方法与定义一般类型的数组方法一样，其一般形式为：

```
struct 结构体名 数组名[数组长度];
```

### 5. 指向结构体变量的指针

定义一个指向结构体变量的指针变量的一般形式为：

```
struct 结构体名 *变量名;
```

通过指向结构体变量的指针  $p$ ，也可以引用结构体中的成员。有以下两种方法：

```
(*p).成员名;
```

和

```
p->成员名;
```

这两种方法是等价的，都表示  $p$  指向的结构体变量中的成员。

引入指向结构体变量的指针概念后，再定义结构体变量时，可以通过动态存储分配函数为结构体变量分配内存空间直接用一个指针指向它，而不一定必须指定变量名。

### 6. 联合的定义

联合 (union) 又叫共用体，是 C 语言中定义的一种构造的数据类型。它利用覆盖技术，将几种不同类型的变量存放到同一段内存单元中。



## 7. 定义联合类型变量的三种方法

(1) 先定义联合类型，再定义联合变量。

```
union 联合名  
{成员表列;  
};  
union 联合名 联合变量 1, 联合变量 2...
```

(2) 在定义联合类型的同时定义联合变量。

```
union 联合名  
{  
成员表列  
}变量名表列;
```

(3) 直接定义联合变量。

```
union  
{  
成员表列  
}变量名表列;
```

## 8. 联合变量的引用

联合变量引用的一般形式为：

```
变量名.成员名
```

## 9. 使用联合类型变量时的注意事项

- (1) 联合变量中起决定作用的是最后一次赋值。
- (2) 联合变量的地址与其成员变量的地址是一样的。
- (3) 不能直接给联合变量赋值。
- (4) 联合变量不能作为函数的参数，函数的返回值也不能是联合类型。

## 10. 用 typedef 定义数据类型

typedef 定义的一般形式为：

```
typedef 原类型名 新类型名
```

其中“原类型名”中含有定义部分，“新类型名”一般用大写表示，以便于区别。

typedef 不但可以用来定义基本类型，还可以用来定义数组、指针、结构等构造类型。用 typedef 定义类型与宏定义不同，前者是在编译时完成，后者是由预处理完成的。

本章讨论 C 语言中的位运算。位运算是 C 语言的一个特色，是 C 语言有别于其他高级语言的主要特征之一。本章将具体介绍 C 语言的 6 种位运算、位运算的规则以及位段的相关知识。

## 8.1 概述

在第 1 章中讲过，C 语言虽然是一类高级语言，但它既具有高级语言的特点又具有低级语言的功能。严格地讲，它是介于高级语言与低级语言之间的一门语言，因此 C 语言具有强大的生命力和广泛的应用领域。C 语言不但可以编写应用软件，还可以用于开发系统软件。

C 语言之所以有如此强大的功能，就是因为在 C 语言中引入了指针和位运算功能。这也是其他高级语言不常有的，它使得 C 语言可以进行一些更为底层的操作。

位运算是指操作的对象是二进制的运算。前面介绍的各种运算都是以字节作为最基本位进行的。但有时这种以字节作为最基本位的操作不能满足程序设计的要求，而常要求在位（bit）一级进行运算或处理。特别是在开发一些系统软件时，这种情况更为明显。C 语言提供的位运算功能就可以深入到位（bit）一级进行运算处理。因此，C 语言的功能更为强大，也能像汇编语言一样用来编写系统程序。

## 8.2 位运算符

C 语言的运算符十分丰富，共有 34 种。在第 2 章中讨论了一些基本运算符的用法，本节主要讨论位运算符。位运算符是帮助进行位一级操作、运算的运算符。C 语言中提供了 6 种位运算符，如表 81 所示。

表 8-1 位运算符

位运算符	解释
&	按位与
^	按位异或
	按位或
~	取反
<<	左移
>>	右移



在这 6 种位运算符中,除了取反运算为单目运算外,其余的运算都为二目运算。另外,上述的 6 种位运算符只能对整型数据或字符型数据进行操作,不能对浮点型数据等进行操作。

### 8.2.1 按位与运算 (&)

按位与运算符是二目运算符,要求运算符两侧各有一个运算量。运算的规则是:

- (1) 参与运算的两数对应的二进位相与。
- (2) 如果两个相应的二进位均为 1,则该两位的按位与结果为 1。
- (3) 否则,该两位的按位与结果为 0。

例如:  $8 \& 10$  可写算式如下:

0000000000001000	(8 的二进制补码)
<u>&amp; 0000000000001010</u>	(10 的二进制补码)
0000000000001000	(8 的二进制补码)

因此  $8 \& 10 = 8$ 。

这里要注意一点,在计算机中,整数是以二进制补码形式存储的,因此两个整数按位与时(或进行其他任何位运算时),都是两数的补码在做按位与(或其他位运算)。

例如:  $(-8) \& (-10)$  可写算式如下:

1111111111111000	(-8 的二进制补码)
<u>&amp; 1111111111110110</u>	(-10 的二进制补码)
1111111111110000	(-16 的二进制补码)

因此  $(-8) \& (-10) = -16$ 。

按位与运算有其特殊功能。

#### (1) 清零功能

显然,无论怎样一个数,只要使它与 0 按位与,就可以将该数各位清零。

例如:

0000000001101000
<u>&amp; 0000000000000000</u>
0000000000000000

#### (2) 取一个数中的某些指定位

例如要取整数 341 的后 8 位(低字位),只需将 341 按位与上 255 即可。

00000001101010101
<u>&amp; 0000000011111111</u>
00000000101010101

显然,按位与后保留了 341 的后 8 位,341 的高 8 位清零。因此,要想将某一个数的某一位保留下来,就与一个数进行按位与运算,此数在要保留的位上取 1,其余位取 0。



## 8.2.2 按位或运算 (|)

按位或运算符是二目运算符，要求运算符两侧各有一个运算量。运算的规则是：

- (1) 参与运算的两数对应的二进制位相或。
- (2) 如果两个相应的二进制位中有一个位为 1，则该两位按位或的结果为 1。
- (3) 否则若两个位全为 0，则该两位的按位或结果为 0。

例如：9|5 可写算式如下：

```
00001001
|00000101
00001101      (十进制为 13)
```

因此，9|5=13。

按位或运算也有其特殊功能，即对一个数据的某些位置 1。

例如将整数 341 的后 8 位（低字位）全置为 1。

```
00000001101010101
|0000000011111111
0000000111111111
```

这样就将 341 的二进制位表示的后 8 位全部置成 1 了。因为 341 的高 8 位与 0 相或，因此高 8 位保持原样。

## 8.2.3 按位异或运算 (^)

按位异或运算也是双目运算，要求运算符两侧各有一个运算量。运算的规则是：

- (1) 参与运算的两数对应的二进制位相异或。
- (2) 如果两个相应的二进制位相异（例如 1^0），则该两位按位异或的结果为 1；
- (3) 否则若两个位相同（例如 1^1），则该两位的按位异或结果为 0。

例如 9^5 可写成算式如下：

```
00001001
^00000101
00001100      (十进制为 12)
```

因此，9^5=12。

按位异或时，两数的对应位分别进行异或运算。两位相异（即两位不同），异或结果为 1，两位相同，异或结果为 0。

按位异或运算有其特殊功能。

- (1) 翻转特定位

例如将二进制表示的整数 0010101110111101 的低字位（后 8 位）按位翻转，就可以将



该数与二进制数 0000000011111111 进行按位异或运算。

```

0010101110111101
^ 0000000011111111
0010101110100010

```

这样就可以将后 8 位按位翻转了。因为任意一位二进制数 (0 或 1) 与 1 异或时, 都要发生翻转, 即  $1^1=0$ ;  $0^1=1$ , 与 0 异或时, 原位不变;  $0^0=0$ 。所以要使二进制数中哪几位翻转, 就将其与进行异或运算的那几位置为 1, 不想改变的那几位置为 0 即可。

## (2) 交换两个值

假设有两个变量  $a=1$ ,  $b=5$ , 现在想将  $a$ 、 $b$  两值互换, 一般的做法是:

```

t=a;
a=b;
b=t;

```

显然, 这种做法一定需要一个临时变量  $t$  作为中介。如果巧妙地运用按位异或运算的方法, 就可以省掉这个临时变量, 从而减少了程序运算的空间复杂度。具体的做法是:

```

a=a^b;
b=b^a;
a=a^b;

```

同样可以达到  $a$ 、 $b$  两值互换的目的。可以通过下面的式子来解释:

$a=a^b$ :

```

000001
^ 000101
000100

```

这一步:  $a=000100$

$b=b^a$ :

```

000101
^ 000100
000001

```

这一步:  $b=000001$  (b 变为 1)

$a=a^b$ :

```

000100
^ 000001
000101

```

这一步:  $a=000101$  (a 变为 5)

显然,  $a$  与  $b$  的值发生了交换。但这只是一个具体实例的描述, 并不能作为证明。下面给出具体的证明。

因为：令  $a' = a \oplus b$ ；（ $a'$ 为中间值），所以： $b = b \oplus a'$ ；即  $b = b \oplus (a \oplus b) = b \oplus a \oplus b = b \oplus b \oplus a = 0 \oplus a = a$   
 同理： $a = a \oplus b = a' \oplus a = a \oplus b \oplus a = a \oplus a \oplus b = 0 \oplus b = b$ 。

因此实现了  $a$  与  $b$  值的交换。

在这个证明中，用到了异或运算的结合律和交换律。

### 8.2.4 取反运算（~）

取反运算是位运算中唯一的单目运算，即只需要一个操作数。运算的规则是：将参加运算的数按位求反，即将 0 变为 1，将 1 变为 0。

例如  $\sim 9$  的运算为：

$\sim(0000000000001001)$  结果为：111111111110110

按位取反运算的一个最好的应用实例就是将计算机中补码表示的数取相反数。看下面这个实例。

#### 例程 8-1 应用取反位运算求相反数。

```
#include <stdio.h>
int main(void)
{
    int a;
    a = ~10 + 1;
    printf("%d", a);
    getchar();
    return 0;
}
```

本例程是应用取反位运算求整数 10 的相反数，即将整数 10 按位取反，末位加 1。本例程的运行结果为：

```
-10
```

之所以可以这样计算一个数的相反数，是跟计算机中数制表示有关的。在 Intel 80X86 系列的计算机中，整数是用 16 位补码表示的。其补码的表示范围是： $-2^{15} \sim 2^{15}-1$ ，补码之间存在着这样的关系：将一个数的补码按位取反，末位加 1 就得到这个补码数的相反数。涉及到计算机中数制表示的相关知识，有兴趣的读者可参看计算机组成原理或计算机组织结构等书目。

### 8.2.5 左移运算（<<）

左移运算是双目运算，要求运算符两侧各有一个运算量。它的作用是将一个数的各二进位全部左移指定的位数。例如：

```
a = a << 2;
```

意思是将变量  $a$  的各二进位全部左移 2 位，右面补 0。

在二进制运算中，左移 1 位相当于将该数乘 2。例如：

二进制数  $(00001111)_2 = 15$ 。

将它左移 1 位得  $(00001111)_2 \ll 1 = (00011110)_2 = 30$ 。

依此类推，左移 2 位就相当于将原数乘  $2^2 = 4$ ，左移 3 位相当于将原数乘  $2^3 = 8$ ，……。



但有时也会出现不符合这个规律的情况，看下面这个例子。

例程 8-2 左移运算的高位溢出。

```
#include <stdio.h>
int main(void)
{
    unsigned a,b;
    a= 32768;
    b=16384;
    a=a<<1;
    b=b<<1;
    printf("32768<<1=%u\n",a);
    printf("16384<<1=%u\n",b);
    getchar();
    return 0;
}
```

本例程中分别将无符号整型数 32 768 和 16 384 左移 1 位，按照前面所讲的那样，应该相当于将两数分别乘 2，但是本程序的运行结果为：

```
32768<<1=0
16384<<1=32768
```

无符号整型数 32 768 左移 1 位后并不是想象的那样扩大 2 倍，而是得 0。产生这种现象的原因就是左移运算的高位溢出。

前面已经讲到，在计算机中，数是以二进制形式存储的。对于无符号整型数而言，任何位都是数值位而没有符号位，无符号整型数 32 768 和 16 384 在计算机内部的存储形式如图 8-1 所示。

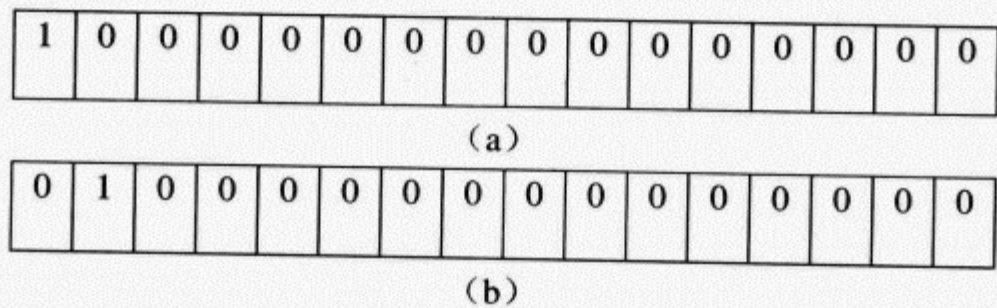


图 8-1 无符号整型数在计算机内部的存储形式

图 8-1 中，(a) 表示 32 768 在计算机内部的存储形式，(b) 表示 16 384 在计算机内部的存储形式。当对这两个数做左移 1 位的操作后，它们分别变为如图 8-2 所示的形式。

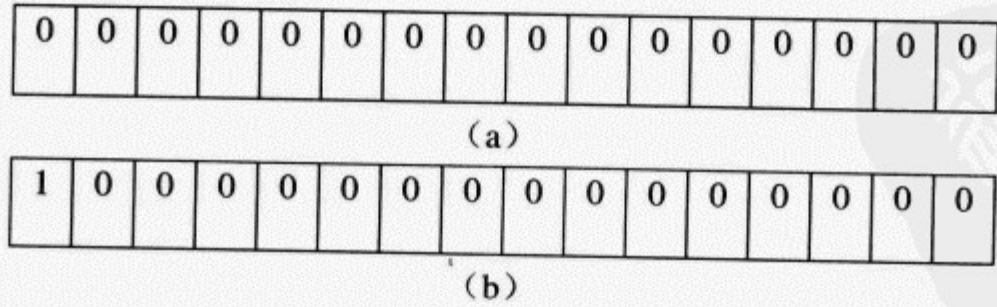


图 8-2 左移 1 位后在计算机内部的存储形式

当 32 768 左移 1 位后，最高位的 1 溢出，右边补 0，这样在计算机内部就是 0 的表示形式了，因此，32 768 左移 1 位后结果为 0。而 16 384 左移 1 位后，最高位 0 溢出（不包

含1)，这样在计算机内部就是32768的表示形式了。因此，16384左移1位后相当于扩大2倍。其实，在第2章中就曾提到，无符号整型数的范围是0~65535，而32768的2倍为65536，并不在无符号整型数表示的范围内，所以这种情况得到的结果必然不是想象的那样扩大2倍，在编程时一定要注意这一点。

因为左移运算较乘法运算快得多，因此在编程时可以灵活运用左移运算实现乘2的功能，这样程序的执行效率会更高。

### 8.2.6 右移运算(>>)

与左移运算类似，右移运算也是双目运算，要求运算符两侧各有一个运算量。它的作用是将一个数的各二进制位全部右移指定的位数。例如：

```
a=a>>2;
```

意思是将变量a的各二进制位全部右移2位，左面补0或1。左面补0还是补1，这取决于所用的计算机系统。有的系统补0，称为“逻辑右移”，有些系统根据原来最高位的情况移入不同的值。如果原数最高位为0，则右移后左面补0；如果原数最高位为1，则右移后左面补1。这种右移称为“算术右移”。Turbo C编译环境采用的是算术右移。

右移操作后左面补0还是补1关系到数的正负。对于正数右移后左面应当补0，这样还保持是正数，对于负数右移后左面应当补1，这样还保持是负数。

在二进制运算中，右移1位相当于将该数除以2。例如：

二进制数 $(00001110)_2=14$ 。

将它右移1位得 $(00001110)_2>>1=(00000111)_2=7$ 。

依此类推，右移2位就相当于将原数除以 $2^2=4$ ，右移3位相当于将原数除以 $2^3=8$ ，……。但有时也会出现不符合这个规律的情况，看下面这个例子。

#### 例程 8-3 右移运算的舍入误差。

```
#include <stdio.h>
int main(void)
{
    int a,b;
    a= -16;
    b=15;
    a=a>>1;
    b=b>>1;
    printf("-16>>1=%d\n",a);
    printf("15>>1=%d\n",b);
    getchar();
    return 0;
}
```

本例程中将整型数-16和15各位右移1位，按照前面所讲的那样，应该相当于将两数分别除以2，但是本程序的运行结果为：

```
-16>>1=-8
15>>1=7
```

其原因就是产生了舍入误差。还是通过整数在计算机内部的二进制存储形式来理解这个问题，整型数-16和15在计算机内部的存储形式如图8-3所示。



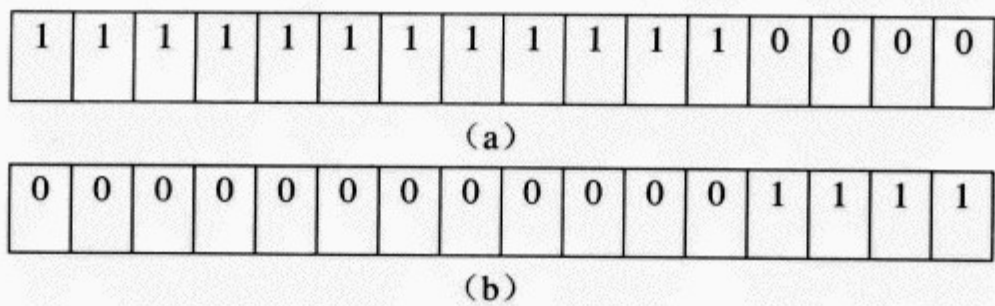


图 8-3 整型数在计算机内部的存储形式

图 8-3 中，(a) 表示-16 在计算机内部的补码形式，(b) 表示整型数 15 在计算机内部的补码形式。当两数分别右移 1 位时，它们分别变为如图 8-4 所示的形式。

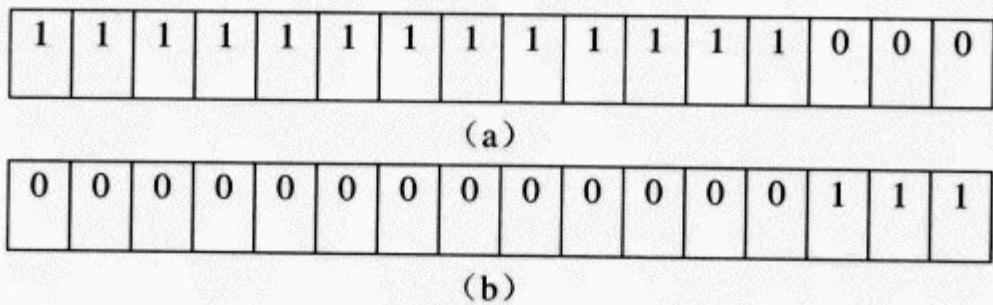


图 8-4 右移 1 位在计算机内部的存储形式

注意，Turbo C 是支持算术右移的，因此-16 右移 1 位后，最高位补 1；15 右移 1 位后，最高位补 0。显然-16 右移 1 位后变为原来的一半，(a) 为-8 的补码表示。而 15 右移 1 位后，由于移出的数值位为 1，低位被自动舍弃，因此 15 右移 1 位后结果为 7。实际上，右移运算中只有当一个数为偶数时，右移 1 位后的结果才相当于原数除以 2；当一个数为奇数时，右移 1 位后的结果必然产生误差。

同样，右移运算较除法运算快得多，因此在编程时可以灵活运用右移运算实现除以 2 的功能，这样程序的执行效率会更高。

### 8.3 位运算中的规则

在进行位运算时要遵守一些规则，下面作简单地介绍。

#### 8.3.1 不同类型数据之间的位运算

在双目运算中，两个不同类型的数据进行位运算时，由于其数据长度可能不同，因此系统会将两数按右端对齐，较短的数据左端补上 0 或 1，然后再进行位运算。如果较短的数据为正数，即最高位为 0，则左端补满 0；如果较短的数据为负数，即最高位为 1，则左端补满 1；如果较短的数据为无符号数，即每一位都是数值位，该数永远为正数，则左端补满 0。

#### 8.3.2 位运算赋值运算符

有时为了方便起见，还可以在编程中使用位运算赋值运算符。例如：`&=`、`^=`、`|=`、`<<=`、`>>=`等。它们的具体含义如表 8-2 所示。



表 8-2 位运算赋值运算符的含义

位运算赋值运算符表达式	含义
a&=b	a=a&b
a^=b	a=a^b
a =b	a=a b
a<<=2	a=a<<2
a>>=2	a=a>>2

8.4 位段

在进行位操作时，有些信息的存储并不需要占用整个字节空间，而只需要其中的几位即可。例如对一个字符进行哈夫曼编码，就是将该字符由 8 位表示压缩成小于 8 位的表示。但 C 语言中没有提供现成的变量存储每一位，为了更最大限度地节省存储空间，C 语言中提供了一种数据结构来帮助解决位存储的问题，这个数据结构叫做“位段”，有些书中也把它称作“位域”。

“位段”就是把一字节中的二进制位划分为几个不同的区段，并说明每个区段的位数。每个段有一个段名，允许在程序中按段名进行操作，这样就可以提取一字节中的几位进行操作了。

8.4.1 位段的定义和位段变量的说明

位段定义的一般形式为：

```
struct 位段结构名
{ 位段列表 };
```

其中位段列表的形式为：

```
类型说明符 位段名: 位段长度
```

例如：

```
struct bitseg {
    int a:8;
    int b:2;
    int c:6;
};
```

就是一个合法的位段类型定义。它表示 struct bitseg 为一个位段类型，里面包含三个位段。位段变量的说明方式与结构变量相同。可先定义后说明，例如：

```
struct bitseg {
    int a:8;
    int b:2;
    int c:6;
};
struct bitseg d;
```

同时定义说明，例如：



```
struct bitseg {
    int a:8;
    int b:2;
    int c:6;
}d;
```

或者直接说明，例如：

```
struct {
    int a:8;
    int b:2;
    int c:6;
}d;
```

这三种方式。

以上三种说明位段变量的方法都是合法的，它们都表示 d 为一个位段类型变量，包含 3 个位段。其中 a 位段占 8 位，b 位段占 2 位，c 位段占 6 位，变量 d 共占用 2 字节。

### 8.4.2 位段的应用举例

下面通过一个例子来理解位段的使用。

#### 例程 8-4 位段的使用。

```
#include<stdio.h>
struct bitseg
{
    unsigned a:1;
    unsigned b:3;
    unsigned c:4;
}
main()
{
    struct bitseg d,*p;
    d.a=1;
    d.b=6;
    d.c=13;
    printf("%d,%d,%d\n",d.a,d.b,d.c);
    p=&d;
    p->a=0;
    p->b=p->b&3;
    p->c=p->c|1;
    printf("%d,%d,%d\n",p->a,p->b,p->c);
}
```

本例程中定义了一个位段结构 bitseg，并声明了一个位段变量 d 和指向 d 的指针 p，然后对位段变量中的每个位段赋值。这里必须注意赋值的大小不能超出位段的表示范围，例如位段 b 只能存储 3 位二进制，因此对 b 的赋值范围为 0~15。然后显示出各位段的值（以整数形式 %d 显示）。这里应该知道，位段的使用和结构成员的使用相同，其一般形式为：

位域变量名·位域名

同时，位段允许用各种格式输出。

P 是一个指向位段变量的指针，它的用法同一般的结构指针用法类似。接下来通过指针 p 引用变量 d 中的各位段，并对 a 位段赋 0，b 位段与 3，c 位段或 1。最后显示进行了位



运算后的各位段的值（仍以整数形式%d显示）。

本例程的运行结果为：

```
1,6,13
0,2,13
```

应用位段的好处是节省了存储空间，实现数据的压缩。本例程中，如果用整型来存储这3个数至少需要6字节大小，然而应用位段存储只需1字节大小。

### 8.4.3 位段的几点说明

下面对位段的使用进行几点说明。

（1）一个位段必须存储在同一字节中，不能跨两字节。如一字节所剩空间不够存放另一位段时，应从下一单元起存放该位段，也可以根据需要使某位段从下一单元开始。

例如：

```
struct bitseg
{
    unsigned a:4
    unsigned :0      /*空段*/
    unsigned b:4      /*从下一单元开始存放*/
    unsigned c:4
}
```

在这个位段定义中，a占第一字节的4位，后4位填0表示不使用；b从第二字节开始，占用4位，c占用4位。

（2）由于位段不允许跨两字节，因此位段的长度不能大于一字节的长度，也就是说不能超过8位二进制。

（3）位段可以无位段名，这时它只用作填充或调整位置。无名的位段是不能使用的。

例如：

```
struct k
{
    int a:1
    int :2      /*该2位不能使用*/
    int b:3
    int c:2
};
```

从以上分析可以看出，位段在本质上就是一种结构类型，不过其成员是按二进制分配的。

## 8.5 本章小结与要点回顾

本章讨论的是位运算。C语言之所以功能强大，既可以编写应用软件，又可以编写系统软件，就是因为在C语言中引入了指针和位运算功能，使得C语言可以进行一些更为底层的操作。下面将本章内容小结。



## 1. 位运算符

C 语言中提供了 6 种位运算符： $\&$ 、 $|$ 、 $\wedge$ 、 $\sim$ 、 $\ll$ 、 $\gg$ 。各自的功能如下。

按位与运算 ( $\&$ ):

- (1) 参与运算的两数对应的二进位相与。
- (2) 如果两个相应的二进位均为 1，则该两位的按位与结果为 1。
- (3) 否则，该两位的按位与结果为 0。

按位或运算 ( $|$ ):

- (1) 参与运算的两数对应的二进位相或。
- (2) 如果两个相应的二进位中有一个位为 1，则该两位按位或的结果为 1。
- (3) 否则若两个位全为 0，则该两位的按位或结果为 0。

按位异或运算 ( $\wedge$ ):

- (1) 参与运算的两数对应的二进位相异或。
- (2) 如果两个相应的二进位相异（例如  $1\wedge 0$ ），则该两位按位异或的结果为 1。
- (3) 否则若两个位相同（例如  $1\wedge 1$ ），则该两位的按位异或结果为 0。

取反运算 ( $\sim$ ):

取反运算是单目运算，只需要一个操作数。其规则是：将参加运算的数按位求反，即将 0 变为 1，将 1 变为 0。

左移运算 ( $\ll$ ):

- (1) 将一个数的各二进位全部左移指定的位数。
- (2) 在二进制运算中，左移 1 位相当于将该数乘 2，依此类推，左移  $n$  位相当于将该数乘  $2^n$ 。
- (3) 左移运算中，该数左移时被溢出舍弃的高位如果全为 0，则符合 (2) 中提到的规律；如果溢出位中包含 1，就没有上述规律。

右移运算 ( $\gg$ ):

- (1) 将一个数的各二进位全部右移指定的位数。
- (2) 右移运算中每右移 1 位，最高位要补 0 或补 1。有的系统补 0，称为“逻辑右移”；有的系统根据原来最高位的情况移入不同的值。如果原数最高位为 0，则右移后左面补 0；如果原数最高位为 1，则右移后左面补 1，这种右移称为“算术右移”。Turbo C 编译环境采用的是算术右移。
- (3) 在二进制运算中，右移 1 位相当于将该数除以 2，依此类推，右移  $n$  位相当于将该数除以  $2^n$ 。
- (4) 右移运算中，该数右移时被溢出舍弃的低位如果全为 0，则符合 (3) 中提到的规律；如果溢出位中包含 1，就没有上述规律。

## 2. 位运算中的规则

在双目运算中两个不同类型的数据进行位运算时，遵循右端对齐、较短数左端补0或补1的规则。

在编程中可以使用位运算赋值运算符。包括： $\&=$ 、 $\wedge=$ 、 $|=$ 、 $\ll=$ 、 $\gg=$  5种。

## 3. 位段

(1) 位段定义的一般形式为：

```
struct 位段结构名  
{ 位段列表 };
```

其中位段列表的形式为：

```
类型说明符 位段名: 位段长度
```

(2) 位段变量的说明方式与结构变量相同。可采用先定义后说明、同时定义说明或者直接说明这三种方式。

(3) 位段的使用和结构成员的使用相同，其一般形式为：

```
位域变量名.位域名
```

位段允许用各种格式输出。

(4) 位段的使用中要注意：位段不能跨两字节；位段的长度不能超出8位；位段可以无位段名，这时它只用作填充或调整位置。



## 第 2 部分

### C 库函数

库函数本身不属于 C 语言，但它是程序开发必不可少的，是每个程序员或者专业人士都必须接触和用到的。本部分主要介绍基于 ANSI C89 标准的 C 库函数，共包括 7 章。第 9 章：C 标准库介绍；第 10 章：I/O 函数；第 11 章：字符处理函数；第 12 章：字符串处理函数；第 13 章：数学函数；第 14 章：时间和日期函数；第 15 章：其他函数。其中第 9 章是对 ANSI C89 标准的 15 个库文件进行总体的介绍，包括每个库文件中定义的宏、类型以及声明的函数。第 10 章~第 15 章则按照函数功能的不同将库函数划分为 6 类，每一章对应阐述了每一类函数的功能、用法，并给出例程解析。

本部分介绍的库函数涵盖了一般 C 程序设计中常用的函数，因此可作为程序设计人员和相关读者的参考手册。



在第 1 章中曾介绍过,美国国家标准协会(American National Standards Institute, ANSI)于 1983 年发表了一个完整的标准 C 语言,通常称之为 ANSI C。ANSI C 标准库中包含了 15 个标准头文件。标准库中的标准函数、类型以及宏分别在这 15 个标准头文件中定义。标准头文件的文件名如表 9-1 所示。

表 9-1 标准库函数

<assert.h>	<float.h>	<math.h>	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h>	<string.h>
<errno.h>	<locale.h>	<signal.h>	<stdio.h>	<time.h>

可以通过以下方式访问头文件:

```
#include<头文件名>
```

或者:

```
#include "头文件名"
```

有关文件包含的知识已在第 5 章中进行了详细的介绍,这里不再赘述。但要注意几点的是:

- (1) 头文件的包含顺序是任意的,可以包含任意多次。
- (2) 头文件一定要包含在任何外部声明和外部定义之外。
- (3) 头文件也可以自己编写,并不一定是标准库中的标准头文件。
- (4) 包含标准头文件时,应当用第一种包含格式,因为标准头文件一般都存放在系统的指定文件目录下。

标准库本身并不是 C 语言的构成部分,它是为开发 C 程序而制定的一种开发环境。因此,支持标准 C 实现的开发环境会提供函数库中的函数声明、类型定义以及宏定义。

本章将对标准库中的<assert.h>、<errno.h>、<float.h>、<limits.h>、<locale.h>、<setjmp.h>、<signal.h>、<stdarg.h>、<stddef.h>这 9 个标准头文件中的内容进行详细地介绍,而对其余 6 个头文件的内容只作简要介绍。因为以上 9 个头文件中定义的宏、类型以及函数在标准库中所占比例很小,内容相对简单,其他 6 个标准头文件中内容所占比例很大,在一般的软件开发中使用相对频繁,由于篇幅限制和出于对内容完整性的考虑,本章只作简要的介绍,但列出其中定义的所有标准函数。

后续章节将按函数的功能,对余下的 6 个头文件的内容进行介绍,除标准函数外还增



加一些常用的非标准函数，这样内容更加丰富，也更加便于用户查阅检索。

## 9.1 诊断：<assert.h>

<assert.h>中只定义了一个带参的宏 `assert`，其定义形式如下：

```
void assert (int 表达式)
```

`assert` 宏用于为程序增加诊断功能，它可以测试一个条件并可能使程序终止。在执行语句 `assert(表达式);`

时，如果表达式为 0，则在终端显示一条信息：

```
Assertion failed: 0, file 源文件名, line 行号
Abnormal program termination
```

然后调用 `abort` 终止程序的执行。

在<assert.h>中，带参宏 `assert` 被定义为条件编译，如果在源文件中定义了宏 `NDEBUG`，则即使包含了头文件<assert.h>，`assert` 宏也将被忽略。

## 9.2 字符类别测试：<ctype.h>

头文件<ctype.h>中定义了一些测试字符的函数，在这些函数中，每个函数的参数都是整型 `int`，而每个参数的值为 `EOF` 或者 `char` 类型的字符。<ctype.h>中定义的标准函数列表如表 9-2 所示。

表 9-2 <ctype.h>中定义的函数

函数定义	函数功能
<code>int isalnum(int c)</code>	检查字符是否是字母或数字
<code>int isalpha(int c)</code>	检查字符是否是字母
<code>int isascii(int c)</code>	检查字符是否是ASCII码
<code>int iscntrl(int c)</code>	检查字符是否是控制字符
<code>int isdigit(int c)</code>	检查字符是否是数字字符
<code>int isgraph(int c)</code>	检查字符是否是可打印字符
<code>int islower(int c)</code>	检查字符是否是小写字母
<code>int isprint(int c)</code>	检查字符是否是可打印字符
<code>int ispunct(int c)</code>	检查字符是否是标点字符
<code>int isspace(int c)</code>	检查字符是否是空格符
<code>int isupper(int c)</code>	检查字符是否是大写字母
<code>int isxdigit(int c)</code>	检查字符是否是十六进制数字字符
<code>int toupper(int c)</code>	将小写字母转换为大写字母
<code>int tolower(int c)</code>	将大写字母转换为小写字母

有关<ctype.h>中定义的这些标准函数以及一些常用的非标准字符处理函数将在第 11



章中进行详细地介绍。

9.3 错误处理: <errno.h>

<errno.h>中定义了两个常量，一个变量。

1. EDOM

它表示数学领域的错误代码。

2. ERANGE

它表示结果超出范围的错误代码。

3. errno

这是一个变量，该值被设置成用来指出系统调用的错误类型。

9.4 浮点算术运算常量: <float.h>

<float.h>中定义了与浮点算术运算相关的一些常量。下面给出这些常量的字符表示以及含义，给出的每个值代表相应量的最小取值，各个实现可以定义适当的值，如表 9-3 所示。

表 9-3 <float.h>中定义的字符常量

字符常量	取值	含义
FLT_RADIX	2	指数表示的基数，例如2、16
FLT_ROUNDS		加法的浮点舍入模式
FLT_DIG	6	表示精度的十进制数字
FLT_EPSILON	1E-5	最小的数x，x满足x+1.0≠1.0
FLT_MANT_DIG		尾数中的数（以FLT_RADIX为基数）
FLT_MAX	1E+37	最大的浮点数
FLT_MAX_EXP		最大的数n，n满足FLT_RADIX <sup>n-1</sup> 仍可表示
FLT_MIN	1E-37	最小的浮点数
FLT_MIN_EXP		最小的数n，n满足10 <sup>n</sup> 是一个规格化数
DBL_DIG	10	表示精度的十进制数字
DBL_EPSILON	1E-9	最小的数x，x满足x+1.0≠1.0
DBL_MANT_DIG		尾数中的数（以FLT_RADIX为基数）
DBL_MAX	1E+37	最大的双精度浮点数
DBL_MAX_EXP		最大的数n，n满足FLT_RADIX <sup>n-1</sup> 仍可表示
DBL_MIN	1E-37	最小的规格化双精度浮点数
DBL_MIN_EXP		最小的数n，n满足10 <sup>n</sup> 是一个规格化数



## 9.5 整型常量: <limits.h>

头文件<limits.h>中定义了一些表示整型大小的常量。这些常量的字符表示以及含义如表 9-4 所示。

表 9-4 <limits.h>中定义的字符常量

字符常量	取值	含义
CHAR_BIT	8	char类型的位数
CHAR_MAX	255或127	char类型最大值
CHAR_MIN	0或-127	char类型最小值
INT_MIN	-32 767	int类型最小值
INT_MAX	32 767	int类型最大值
LONG_MAX	2 147 483 647	long类型最大值
LONG_MIN	-2 147 483 647	long类型最小值
SCHAR_MAX	127	signed char类型最大值
SCHAR_MIN	-127	signed char类型最小值
SHRT_MAX	32 767	short类型的最大值
SHRT_MIN	-32 767	short类型的最小值
UCHAR_MAX	255	unsigned char类型最大值
UINT_MAX	65 535	unsigned int类型最大值
ULONG_MAX	4 294 967 295	unsigned long类型最大值
USHRT_MAX	65 535	unsigned short类型的最大值

## 9.6 地域环境: <locale.h>

在<locale.h>中，定义了 7 个常量，1 个结构，2 个函数。

### 1. 常量的定义

- ✧ LC\_ALL: 传递给 setlocale 的第一个参数，指定要更改该 locale 的哪个方面。
- ✧ LC\_COLLATE: strcoll 和 strxfrm 的行为。
- ✧ LC\_CTYPE: 字符处理函数。
- ✧ LC\_MONETARY: localeconv 返回的货币信息。
- ✧ LC\_NUMERIC: localeconv 返回的小数点和货币信息。
- ✧ LC\_TIME: strftime 的行为。

以上扩展成具有唯一取值的整型常数表达式，可作为 setlocale 的第一个参数。

- ✧ NULL: 由实现环境定义的空指针。

### 2. struct lconv 结构

该结构用于存储和表示当前 locale 的设置。其结构定义如下：



```
struct lconv
{
    char *decimal_point;
    char *thousands_sep;
    char *grouping;
    char *int_curr_symbol;
    char *currency_symbol;
    char *mon_decimal_point;
    char *mon_thousands_sep;
    char *mon_grouping;
    char *positive_sign;
    char *negative_sign;
    char int_frac_digits;
    char frac_digits;
    char p_cs_precedes;
    char p_sep_by_space;
    char n_cs_precedes;
    char n_sep_by_space;
    char p_sign_posn;
    char n_sign_posn;
};
```

3. 函数

```
struct lconv *localeconv(void);
```

函数 localeconv 将一个 struct lconv 类型对象的数据成员设置成为按照当前地域环境的有关规则进行数量格式化后的相应值。

```
char *setlocale(int category, char * locale);
```

函数 setlocale 用于更改和查询程序当前的整个地域环境或部分设置。地域环境变量由参数 category（上面定义的 6 个常量）和 locale 指定。

9.7 数学函数: <math.h>

<math.h>中定义了一些数学函数和宏，用来实现不同种类的数学运算。<math.h>中标准数学函数的定义及功能，如表 9-5 所示。

表 9-5 <math.h>中定义的函数

函数定义	函数功能
double exp(double x);	指数运算函数，求e的x次幂函数
double log(double x);	对数函数ln(x)
double log10(double x);	对数函数log
double pow(double x, double y);	指数函数（x的y次方）
double sqrt(double x);	计算平方根函数
double ceil(double x);	向上舍入函数
double floor(double x);	向下舍入函数
double fabs(double x);	求浮点数的绝对值
double ldexp(double x, int n);	装载浮点数函数
double frexp(double x, int* exp);	分解浮点数函数



(续表)

函数定义	函数功能
double modf(double x, double* ip);	分解双精度数函数
double fmod(double x, double y);	求模函数
double sin(double x);	计算x的正弦值函数
double cos(double x);	计算x的余弦值函数
double tan(double x);	计算x的正切值函数
double asin(double x);	计算x的反正弦函数
double acos(double x);	计算x的反余弦函数
double atan1(double x);	反正切函数1
double atan2(double y, double x);	反正切函数2
double sinh(double x);	计算x的双曲正弦值
double cosh(double x);	计算x的双曲余弦值
double tanh(double x);	计算x的双曲正切值

在标准库中，还有一些与数学计算有关的函数定义在其他头文件中。有关<math.h>中定义的标准函数以及其他一些关于数学计算的函数将在第 13 章中详细介绍。

## 9.8 非局部跳转：<setjmp.h>

头文件<setjmp.h>中定义了一种特别的函数调用和函数返回顺序的方式，这种方式不同于以往的函数调用和返回顺序，它允许程序流程立即从一个深层嵌套的函数中返回。

<setjmp.h>中定义了两个宏：

```
int setjmp(jmp_buf env); /*设置调转点*/
```

和

```
longjmp(jmp_buf jmpb, int retval); /*跳转*/
```

宏 setjmp 的功能是将当前程序的状态保存在结构 env 中，为调用宏 longjmp 设置一个跳转点。setjmp 将当前信息保存在 env 中供 longjmp 使用，其中 env 是 jmp\_buf 结构类型的，该结构定义为：

```
typedef struct {
    unsigned j_sp;
    unsigned j_ss;
    unsigned j_flag;
    unsigned j_cs;
    unsigned j_ip;
    unsigned j_bp;
    unsigned j_di;
    unsigned j_es;
    unsigned j_si;
    unsigned j_ds;
} jmp_buf[1];
```

直接调用 setjmp 时，返回值为 0，这一般用于初始化（设置跳转点时）。以后再调用 longjmp 宏时用 env 变量进行跳转，程序会自动跳转到 setjmp 宏的返回语句处，此时 setjmp



的返回值为非 0，由 `longjmp` 的第二个参数指定。

下面通过例子来理解 `<setjmp.h>` 中定义的这两个宏。

### 例程 9-1 非局部跳转演示。

```
#include <setjmp.h>
jmp_buf env; /*定义 jmp_buf 类型变量*/
int main(void)
{
    int value;
    value = setjmp(env); /*调用 setjmp, 为 longjmp 设置跳转点*/
    if (value != 0)
    {
        printf("Longjmp with value %d\n", value);
        exit(value); /*退出程序*/
    }
    printf("Jump ... \n");
    longjmp(env, 1); /*跳转到 setjmp 语句处*/
    return 0;
}
```

本例程先应用 `setjmp` 宏为 `longjmp` 设置跳转点，当第一次调用 `setjmp` 时返回值为 0，并将程序的当前状态（寄存器的相关状态）保存在结构变量 `env` 中。当程序执行到 `longjmp` 时，系统会根据 `setjmp` 保存的状态 `env` 跳转到 `setjmp` 语句处，并根据 `longjmp` 的第二个参数设置此时 `setjmp` 的返回值。

本例程的运行结果为：

```
Jump ...
Longjmp with value 1
```

一般地，宏 `setjmp` 和 `longjmp` 是成对使用的，这样程序流程可以从一个深层嵌套的函数中返回。

## 9.9 信号：<signal.h>

头文件 `<signal.h>` 中提供了一些处理程序运行期间引发的各种异常条件的功能，例如一些来自外部的中断信号等。

在 `<signal.h>` 中只定义了两个函数：

```
int signal(int sig, sigfun fname);
```

和

```
int raise(int sig);
```

`signal` 函数的作用是设置某一信号的对应动作。其中参数 `sig` 用来指定哪一个信号被设置为处理函数。在标准 C 中支持的信号如表 9-6 所示。

参数 `fname` 是一个指向函数的指针，当 `sig` 的信号发生时，程序会自动中断转而执行 `fname` 指向的函数。执行完毕再返回断点继续执行程序。系统提供了两个常量函数指针，可以作为函数的参数传递，分别如下。



表 9-6 标准C支持的信号

取值	说明	默认执行动作	使用的操作系统
SIGABRT	异常中止	中止程序	UNIX DOS
SIGPPE	算术运算错误	中止程序	UNIX DOS
SIGILL	非法硬件指令	中止程序	UNIX DOS
SIGINT	终端中断	中止程序	UNIX DOS
SIGSEGV	无效的内存访问	中止程序	UNIX DOS
SIGTERM	中止信号	中止程序	UNIX DOS

- ✧ SIG\_DEF: 执行默认的系统第一的函数。
- ✧ SIG\_IGN: 忽略此信号。

raise 函数的作用是向正在执行的程序发送一个信号，使当前进程产生一个中断而转向信号处理函数 signal 执行。其中参数 sig 为信号名称，它的取值范围与函数 signal 中参数 sig 取值范围相同，如表 9-6 所示。

下面通过例子理解函数 signal 和 raise。

例程 9-2 signal 和 raise 函数演示。

```
#include <stdio.h>
#include <signal.h>
void Print1();
void Print2();
int main()
{
    signal(SIGINT,Print1);
    printf("Please enter Ctr+c for interupt\n") ;
    getchar();
    signal(SIGSEGV,Print2);
    printf("Please enter any key for a interupt\n");
    getchar();
    raise(SIGSEGV);
}
void Print1()
{
    printf("This is a SIGINT interupt!\n");
}
void Print2()
{
    printf("This is a SIGSEGV interupt!\n");
}
```

本例程首先通过用户终端输入 Ctrl+c 产生一个终端中断，然后应用 signal 函数调用中断处理函数 Print1；再通过 raise 函数生成一个无效内存访问中断，并通过 signal 函数调用中断处理函数 Print2。

本例程的运行结果为：

```
Please enter Ctr+c for interupt
^C
This is a SIGINT interupt!

Please enter any key for a interupt
a
```



```
This is a SIGSEGV interupt!
```

## 9.10 可变参数表: <stdarg.h>

可变参数表<stdarg.h>中的宏用来定义参数可变的函数。在C语言中,有些库函数或用户自定义函数的参数是可变的,常用省略号“.....”(例如库函数中的printf),定义这样的函数就要使用到<stdarg.h>中的宏。

### 1. va\_list

用于保存宏 va\_start、va\_arg 以及 va\_end 所需信息的数据类型。

### 2. <stdarg.h>中还定义了三个宏

```
void va_start(va_list ap, parmN);  
type va_arg(va_list ap, type);  
void va_end (va_list ap);
```

va\_start 的作用是初始化 ap, 因此 va\_start 要在所有其他 va\_开头的宏前面最先使用(除了用 va\_list 定义变量外), 后面的 va\_copy, va\_arg, va\_end 都要用到 ap。在一对 va\_start 和 va\_end 之间不能再次使用 va\_start 宏。其中, parmN 为“...”之前的最后一个参数。例如, printf 函数定义为: printf(const char\*format, ...); 那么在 printf 函数中的 va\_start 使用之后, parmN 的值就等于\*format。

va\_arg 的作用是返回参数列表 ap 中的下一个具有 type 类型的参数, 每次调用 va\_arg 都会修改 ap 的值, 这样才能连续不断地获取下一个 type 类型的参数。

va\_end 与 va\_start 构成了一个 scope, va\_end 标志着结束, 之后的 ap 就无效了。

## 9.11 公共定义: <stddef.h>

在头文件<stddef.h>中, 指定了标准库中的公共定义。其中主要包括以下内容。

### 1. NULL

空指针类型常量。

### 2. offset(type, member-designator)

它是扩展 iz-t 类型的一个整型常数表达式。它的值为从 type 定义的结构类型的开头到结构成员 member-designator 的偏移字节数。

### 3. ptrdiff\_t

表示两指针之差的带符号整数类型。

### 4. size\_t

表示由 sizeof 运算符计算出的结果类型, 它是一个无符号整数类型。



5. wchar\_t

它是一种整数类型，取值范围为在被支持的地域环境中，最大扩展字符集的所有字符的各种代码，空字符代码值为 0。

9.12 输入输出：<stdio.h>

头文件<stdio.h>中定义了输入输出函数、类型和宏、它们几乎占到标准库的三分之一。<stdio.h>中声明的函数以及功能如表 9-7 所示。

表 9-7 <stdio.h>中声明的函数及功能

函数定义	函数功能
FILE *fopen(char *filename, char *type)	打开一个文件
FILE *fopen(char *filename, char *type, FILE *fp)	打开一个文件，并将该文件关联到fp指定的流
int fflush(FILE *stream)	清除一个流
int fclose(FILE *stream)	关闭一个文件
int remove(char *filename)	删除一个文件
int rename(char *oldname, char *newname)	重命名文件
FILE *tmpfile(void)	以二进制方式打开暂存文件
char *tmpnam(char *sptr)	创建一个唯一的文件名
int setvbuf(FILE *stream, char *buf, int type, unsigned size)	把缓冲区与流相关
int printf(char *format...)	产生格式化输出的函数
int fprintf(FILE *stream, char *format[, argument,...])	传送格式化输出到一个流中
int scanf(char *format[,argument,...])	执行格式化输入
int fscanf(FILE *stream, char *format[,argument...])	从一个流中执行格式化输入
int fgetc(FILE *stream)	从流中读取字符
char *fgets(char *string, int n, FILE *stream)	从流中读取一字符串
int fputc(int ch, FILE *stream)	送一个字符串到一个流中
int fputs(char *string, FILE *stream)	送一个字符到一个流中
int getc(FILE *stream)	从流中取字符
int getchar(void)	从stdin流中读取字符
char *gets(char *string)	从流中读取一字符串
int putchar(int ch)	在stdout上输出字符
int puts(char *string)	送一个字符串到流中
int ungetc(char c, FILE *stream)	把一个字符退回到输入流中
int fread(void *ptr, int size, int nitems, FILE *stream)	从一个流中读取数据
int fwrite(void *ptr, int size, int nitems, FILE *stream)	写内容到流中
int fseek(FILE *stream, long offset, int fromwhere)	重定位流上的文件指针
long ftell(FILE *stream)	返回当前文件指针
int rewind(FILE *stream)	将文件指针重新指向一个流的开头
int fgetpos(FILE *stream)	取得当前文件的句柄
int fsetpos(FILE *stream, const fpos_t *pos)	定位流上的文件指针
void clearerr(FILE *stream)	复位错误标志

(续表)

函数定义	函数功能
int feof(FILE *stream)	检测流上的文件结束符
int ferror(FILE *stream)	检测流上的错误
void perror(char *string)	系统错误信息

在头文件<stdio.h>中还定义了一些类型和宏，有关这些函数、类型以及宏的内容将在第 10 章中进行详细介绍。

9.13 实用函数：<stdlib.h>

在头文件<stdlib.h>中声明了一些实现数值转换、内存分配等类似功能的函数。<stdlib.h>中声明的函数以及功能如表 9-8 所示。

表 9-8 <stdlib.h>中声明的函数及功能

函数定义	函数功能
double atof(const char *s)	将字符串s转换为double类型
int atoi(const char *s)	将字符串s转换为int类型
long atol(const char *s)	将字符串s转换为long类型
double strtod (const char*s,char **endp)	将字符串s前缀转换为double类型
long strtol(const char*s,char **endp,int base)	将字符串s前缀转换为long类型
unsinged long strtol(const char*s,char **endp,int base)	将字符串s前缀转换为unsinged long类型
int rand(void)	产生一个0~RAND_MAX之间的伪随机数
void srand(unsigned int seed)	初始化随机数发生器
void *calloc(size_t nelem, size_t elsize)	分配主存储器
void *malloc(unsigned size)	内存分配函数
void *realloc(void *ptr, unsigned newsize)	重新分配主存
void free(void *ptr)	释放已分配的块
void abort(void)	异常终止一个进程
void exit(int status)	终止应用程序
int atexit(atexit_t func)	注册终止函数
char *getenv(char *envvar)	从环境中取字符串
void *bsearch(const void *key, const void *base, size_t *nelem, size_t width, int(*fcmp)(const void *, const *))	二分法搜索函数
void qsort(void *base, int nelem, int width, int (*fcmp)())	使用快速排序例程进行排序
int abs(int i)	求整数的绝对值
long labs(long n)	取长整型绝对值
div_t div(int number, int denom)	将两个整数相除，返回商和余数
ldiv_t ldiv(long lnumer, long ldenom)	两个长整型数相除，返回商和余数

有关上面列出的标准实用函数的功能、用法、例程，将在后续章节中进行详细地介绍。



### 9.14 字符串函数: <string.h>

在头文件<string.h>中定义了一些字符串函数。可以将它们分为两组，第一组函数名以str 开头，主要进行字符串的操作；第二组函数名以 mem 开头，按照字符数组的方式操作对象。<string.h>中声明的函数以及功能如表 9-9 所示。

表 9-9 <string.h>中声明的函数及功能

函数定义	函数功能
char *strcpy(char *str1, char *str2)	串拷贝函数
char *strncpy(char *d, char *s, int m)	串拷贝函数
char *strcat(char *destin, char *source)	字符串拼接函数
char *strncat(char *d, char *s,int n)	字符串拼接函数
int strcmp(char *str1, char *str2)	串比较函数
int strncmp(char *str1, char *str2, int m)	串比较函数
char *strchr(char *str, char c)	在一个串中查找给定字符的第一个匹配处
char *strrchr(char *str, char c)	在串中查找指定字符的最后一个出现处
size_t strspn(char *str1, char *str2)	在串中查找指定字符集的子集的第一次出现
size_t strcspn(char *str1, char *str2)	在串中查找第一个给定字符集内容的段
char *strpbrk(char *str1, char *str2)	在串中查找给定字符集中的字符
char *strstr(char *str1, char *str2)	在串中查找指定字符串的第一次出现
size_t strlen(char *cs)	求字符串的长度
char *strerror(int errnum)	返回指向错误信息字符串的指针
char *strtok(char *str1, char *str2)	查找由在第二个串中指定的分界符分隔开的单词
void *memcpy(void *d, void *s, unsigned n)	从源s中复制n字节到目标d中
void *memmove(void *d, void *s, unsigned n)	移动一字节
void *memcmp(void *s1, void *s2, unsigned n )	比较两个串s2和s1的前n字节
void *memchr(void *s, char ch, unsigned n)	在数组的前n字节中搜索字符
void *memset(void *s, char ch, unsigned n)	将s的前n个字符替换为字符ch

有关头文件<string.h>中声明的函数以及一些常用的字符串函数将在第 12 章中进行详细介绍。

### 9.15 日期与时间函数: <time.h>

在头文件<time.h>中声明了一些处理日期和时间的类型与函数。clock\_t 和 time\_t 是两个表示时间值的算术类型。结构 struct tm 存储了日历时间的各个成分。结构 tm 成员的意义及其正常的取值范围如下。

```
struct tm {
    int tm_sec;      /*从当前分钟开始经过的秒数(0,61)*/
    int tm_min;      /*从当前小时开始经过的分钟数(0,59)*/
    int tm_hour;     /*从午夜开始经过的小时数(0,23)*/
    int tm_mday;     /*当月的天数(1,31)*/
```



```
int tm_mon;      /*从 1 月起经过的月数(0,11)*/
int tm_year;     /*从 1900 年起经过的年数*/
int tm_wday;     /*从本周星期天开始经过的天数(0,6)*/
int tm_yday;     /*从今年 1 月 1 日起经过的天数(0,356)*/
int tm_isdst;    /*夏令时标记*/
};
```

如果夏令时有效，tm\_isdst 值为正；若夏令时无效，tm\_isdst 值为 0；如果得不到夏令时信息，tm\_isdst 值为负。

<time.h>中声明的时间函数如表 9-10 所示。

表 9-10 <time.h>中声明的时间函数

函数定义	函数功能
clock_t clock(void)	确定处理器时间函数
time_t time(time_t *tp)	返回当前日历时间
double difftime(time_t time2, time_t time1)	计算两个时刻之间的时间差
time_t mktime(struct tm *tp)	将分段时间值转换为日历时间值
char *asctime(const struct tm *tblock)	转换日期和时间为ASCII码
char *ctime(const time_t *time)	把日期和时间转换为字符串
struct tm *gmtime(const time_t *timer)	把日期和时间转换为格林尼治标准时间（GMT）
struct tm *localtime(const time_t *timer)	把日期和时间转变为结构
size_t strftime(char *s,size_t smax,const char *fmt, const struct tm *tp)	根据fmt的格式要求将*tp中的日期与时间转换为指定格式

有关头文件<time.h>中声明的函数将在第 14 章中进行详细介绍。





## I/O 函数

I/O 函数也叫输入输出函数，是 C 标准库函数中十分重要的一类函数。所有 I/O 函数都定义在<stdio.h>的头文件中。因此，如果要在程序中调用 I/O 函数，就必须将头文件<stdio.h>包含到源程序文件中，如下所示：

```
#include <stdio.h>
```

或者

```
#include "stdio.h"
```

只有包含了头文件，才可以正常地使用库中的 I/O 函数。头文件<stdio.h>中定义的输入输出函数、类型以及宏的数目几乎占据了整个标准库的三分之一，足见 I/O 函数的重要。

要想使用好输入输出函数，首先要对文件的概念有所了解。本章先对文件做一个简要的概述，然后对 I/O 函数进行详细地介绍。

### 10.1 文件概述

在程序设计中，文件是十分重要的概念，它是存储在外部设备上的数据集合。一般地，操作系统以文件为单位，在外设（磁盘介质）和内存之间对数据进行管理。在读取文件时，系统首先按照用户提供的文件名找到该文件在磁盘上的位置，然后读取数据。同样，在输出文件时，系统也要首先按照用户提供的文件名在磁盘上创立文件，然后把文件数据写入。因此可以看出，文件名在整个文件操作中是非常重要的。在 C 语言中，设置一个文件类型指针指向要操作的文件，通过该指针，用户可以很方便地对文件进行操作。接下来要介绍的所有 I/O 函数都是基于这个指针的，也就是说，以这个指针为参数，对文件进行各种操作。在<stdio.h>中，文件类型的声明如下：

```
typedef struct {
    short          level;          /*缓冲区空满程度*/
    unsigned       flags;          /*文件状态标志*/
    char           fd;             /*文件描述符*/
    unsigned char  hold;           /*如无缓冲区不读取字符*/
    short          bsize;          /*缓冲区大小*/
    unsigned char  *buffer;        /*数据缓冲区指针*/
    unsigned char  *curp;          /*指针，当前指向*/
    unsigned       istemp;         /*临时文件指示器*/
    short          token;          /*有效性检测*/
} FILE;
```

这样定义一个文件时，就可以直接应用 FILE 类型定义文件指针，用以存放文件数据了。

```
FILE *fp;
```

`fp` 指向 `FILE` 类型变量，它可以理解为指向文件类型的指针变量。但实际上，`fp` 并不像其他指针型变量那样指向实际的文件，而是指向 `FILE` 类型的一个结构。在计算机内部，一个 `FILE` 类型的结构对应磁盘中一个真实的文件，从 `FILE` 的定义中看出，在结构中描述了对应文件的具体信息。因此，通过 `fp` 这个指针，用户就可以方便地控制 `FILE` 类型变量对应的那个文件。

## 10.2 clearerr 复位错误标志函数

函数原型：void clearerr(FILE \*fp);

头文件：#include <stdio.h>

是否是标准函数：是

函数功能：复位错误标志，使 `fp` 所指向的文件中的错误标志和文件结束标志置 0。当输入输出函数对文件进行读写出错时，文件就会自动产生错误标志，这样会影响程序对文件的后续操作。`clearerr` 函数就是要复位这些错误标志，也就是使 `fp` 所指向的文件的错误标志和文件结束标志置 0，从而使文件恢复正常。

返回值：无

### 例程 10-1 复位错误标志演示。

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    char ch;
    /* 以写的方式打开一个文件名为 test.txt 的文件 */
    fp = fopen("test.txt", "w");
    /* 错误地从 fp 所指定的文件中读取一个字符，并打印它 */
    ch = fgetc(fp);
    if (ferror(fp))
    {
        /* 如果此操作错误，就发布错误信息 */
        printf("This is a error reading!\n");
        /* 复位错误标志 */
        clearerr(fp);
    }
    /* 关闭文件 */
    fclose(fp);
    return 0;
}
```

例程说明：

(1) 首先程序以只写的方式打开一个文件名为 `test.txt` 的文件。这样，该文件就只能写而不能读了。

(2) 程序企图应用 `fgetc` 函数从 `fp` 所指的文件中读取一个字符，这是违法的，因此文件自动产生错误标志。

(3) 当用 `ferror` 函数检测出文件流存在错误时，就发布一条错误信息，并用 `clearerr`



函数清除 fp 指定的文件流所使用的错误标志，也就是使 fp 所指的文件的错误标志和文件结束标志置 0。这样原先的错误就不会对文件的后续操作产生影响。

注意：ferror 函数与 clearerr 函数应该配合使用，通过 ferror 函数检测出文件有错误标志后要用 clearerr 函数复位错误标志。

### 10.3 fopen、fclose 文件的打开与关闭函数

函数原型：FILE \*fopen(char \*filename, char \*type);  
int fclose(FILE \*fp);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：函数 fopen 打开一个流，即打开一个文件。该函数有两个参数，filename 是需要打开文件的文件名，type 是打开文件的方式。函数 fclose 关闭一个流，即关闭一个文件，并释放文件缓冲区。fclose 函数与 fopen 函数是相对的两个函数。fclose 函数的参数是指向文件的指针，该函数的作用是在程序结束之前关闭文件，并释放文件缓冲区，这样可以保证文件的数据不流失。

在这里，特别列出所有的文件打开方式，以供大家参考，如表 10-1 所示。

表 10-1 文件的打开方式

文件使用方式	意义
r	只读打开一个文本文件，只允许读数据
w	只写打开或建立一个文本文件，只允许写数据
a	追加打开一个文本文件，并在文件末尾写数据
rb	只读打开一个二进制文件，只允许读数据
wb	只写打开或建立一个二进制文件，只允许写数据
ab	追加打开一个二进制文件，并在文件末尾写数据
r+	读写打开一个文本文件，允许读和写
w+	读写打开或建立一个文本文件，允许读写
a+	读写打开一个文本文件，允许读，或在文件末追加数据
rb+	读写打开一个二进制文件，允许读和写
wb+	读写打开或建立一个二进制文件，允许读和写
ab+	读写打开一个二进制文件，允许读，或在文件末追加数据

返回值：fopen 返回 FILE 类型，如果打开的文件存在，返回指向该文件的指针；如果打开的文件不存在，则在指定的目录下建立该文件并打开，并返回指向该文件的指针。Fclose 返回整型，有错返回非 0，否则返回 0。

**例程 10-2** 打开并输出一个文件，然后关闭。

```
#include <string.h>
#include <stdio.h>
int main(void)
```



```
{
FILE *fp;
char buf[11] = "abcdefghij";
/* 以写方式打开文件名为 test.txt 的文件 */
fp = fopen("test.txt", "w");
/*把字符串写入文件中*/
fwrite(&buf, strlen(buf), 1, fp);
/* 关闭文件 */
fclose(fp);
return 0;
}
```

#### 例程说明:

(1) 首先开辟一个 11 字节大小的缓冲区 buf, 也就是数组, 但预先只能存入 10 个字符。这是因为 C 语言中规定数组存放字符串时, 最后一字节要以 '\0' 作为结束标志, 并由系统自动在字符串末尾添加 '\0'。因此, 11 字节大小的缓冲区只存放 10 字节长度的字符串。

(2) 用 fopen 函数以写的方式打开一个名为 test.txt 的文件并将字符串写入文件。调用 fclose 函数关闭该文件。

(3) fclose 函数与 fopen 函数正好相对, 其作用是关闭一个文件。当使用 fopen 函数打开一个文件时, 会返回一个指向该文件的指针。在该例程中这个指针被赋值给 fp, 也就是说 fp 指向了 test.txt 这个文件。而当调用 fclose 函数关闭该文件时, fp 就不再指向该文件了, 相应的文件缓冲区也被释放。

---

注意: 用户在编写程序时应该养成及时关闭文件的习惯, 如果不及时关闭文件, 文件数据有可能会丢失。

---

## 10.4 feof 检测文件结束符函数

函数原型: int feof(FILE \*fp);

头文件: #include<stdio.h>

是否是标准函数: 是

函数功能: 检测流上的文件结束符, 即检测文件是否结束。应用该函数可以判断一个文件是否到了结尾, 此外在读取一个未知长度文件时, 这个函数很有用。

返回值: 遇到文件结束符返回非 0, 否则返回 0。

#### 例程 10-3 检测文件结束标志演示。

```
#include <stdio.h>
int main(void)
{
FILE *stream;
/*以只读方式打开 test.txt 文件*/
stream = fopen("test.txt", "r");
/*从文件中读取一个字符*/
fgetc(stream);
/*检测是否是 EOF, 即结束标志*/
if (feof(stream))
```



```

    printf("Have reached the end of the file!\n");
    /*关闭该文件*/
    fclose(stream);
    return 0;
}

```

#### 例程说明:

- (1) 首先程序打开一个名为 test.txt 的文件。
- (2) 应用 fgetc 函数从一个名为 test.txt 的文件中读取一个字符。
- (3) 判断它是否为文件结束标志 EOF。如果是,说明该文件已经结束,于是在屏幕上显示一条提示信息;如果不是,说明文件还未结束,信息不显示。
- (4) 最后关闭文件。

注意:在实际应用中,feof 函数很重要,利用它程序员就可以很方便地判断当前的文件是否结束,从而进行不同的处理。例如,在从一个未知长度的文件中读取信息时,就可以利用 feof 函数判断什么时候该文件读完。

## 10.5 ferror 检测流上的错误函数

函数原型: int ferror(FILE \*fp);

头文件: #include <stdio.h>

是否是标准函数: 是

函数功能: 检测流上的错误。即检查文件在使用各种输入输出函数进行读写时是否出错。当输入输出函数对文件进行读写时出错,文件就会产生错误标志,应用这个函数,就可以检查出 fp 所指向的文件操作是否出错。

返回值: 未出错返回值为 0,有错返回值非 0。

#### 例程 10-4 应用 ferror 函数检查流上的错误。

```

#include <stdio.h>
int main(void)
{
    FILE *fp;
    char ch;
    /*以写的方式打开一个文件名为 test.txt 的文件 */
    fp = fopen("test.txt", "w");
    /*错误地从 fp 所指定的文件中读取一个字符,并打印它*/
    ch = fgetc(fp);
    printf("%c\n",ch);
    if (ferror(fp))
    {
        /*如果此操作错误,就发布错误信息*/
        printf("Error reading from test.txt !\n");
        /*复位错误标志*/
        clearerr(fp);
    }
    /*关闭文件*/
    fclose(fp);
    return 0;
}

```



**例程说明:**

(1) 首先以只写的方式打开一个文件名为 test.txt 的文件, 这样, 该文件就只能写而不能读了。程序企图用 fgetc 函数从 fp 所指的文件中读取一个字符, 这是非法的操作, 也就是说在用 fgetc 函数进行读取字符时出错了, 因此文件产生错误标志。

(2) 再用 ferror 函数来检测输入输出函数进行文件读写操作时是否出错, 结果发现有错, 因此函数返回一个非 0 整型数, 并提示出错信息。

## 10.6 fflush 清除文件缓冲区函数

**函数原型:** int fflush(FILE \*fp);

**头文件:** #include <stdio.h>

**是否是标准函数:** 是

**函数功能:** 清除一个流, 即清除文件缓冲区, 当文件以写方式打开时, 将缓冲区内容写入文件。也就是说, 对于 ANSI C 规定的缓冲文件系统, 函数 fflush 用于将缓冲区的内容输出到文件中去。

**返回值:** 如果成功刷新, fflush 返回 0; 指定的流没有缓冲区或者只读打开时也返回 0 值; 返回 EOF 指出一个错误。

### 例程 10-5 第一种方式读写文件。

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <io.h>

int main(void)
{
    FILE *stream1, *stream2;
    char test[20] = "This is a test";
    char res[20];
    /*以写的方式打开文件 test.txt*/
    stream1 = fopen("test.txt", "w");
    /*向文件写入字符串*/
    fwrite(test, 15, 1, stream1);
    /*以读的方式打开文件 test.txt*/
    stream2 = fopen("test.txt", "r");
    /*将文件内容读入缓冲区*/
    if(fread(res, 15, 1, stream2))
        printf("%s", res);
    else
        printf("Read error!\n");
    fclose(stream1);
    fclose(stream2);
    getch();
    return 0;
}
```

### 例程 10-6 第二种方式读写文件。

```
#include <string.h>
#include <stdio.h>
```



```
#include <conio.h>
#include <io.h>

int main(void)
{
    FILE *stream1,*stream2;
    char test[20]="This is a test";
    char res[20];
    /*以写的方式打开文件 test.txt*/
    stream1 = fopen("test.txt", "w");
    /*向文件写入字符串*/
    fwrite(test,15,1,stream1);
    /*将缓冲区的内容写入文件*/
    fflush(stream1);
    /*以读的方式打开文件 test.txt*/
    stream2 = fopen("test.txt", "r");
    /*将文件内容读入缓冲区*/
    if(fread(res,15,1,stream2))
        printf("%s",res);
    else
        printf("Read error!\n");
    fclose(stream1);
    fclose(stream2);
    getch();
    return 0;
}
```

#### 例程说明:

例程 10-5 中定义了两个文件指针 stream1 和 stream2。

(1) 首先以写的方式打开文件 test.txt, 用指针 stream1 指向该文件, 并向文件中写入字符串 "This is a test"。

(2) 不关闭该文件, 以读的方式打开文件 test.txt, 并用指针 stream2 指向该文件, 试图将刚刚写入的字符串读入到内存缓冲区中。如果读入成功, 打印出该字符串, 否则报错。

实践证明, 例程 10-5 的输出结果是在屏幕上显示错误信息 Read error!。

例程 10-6 中定义了两个文件指针 stream1 和 stream2。

(1) 首先以写的方式打开文件 test.txt, 用指针 stream1 指向该文件, 并向文件中写入字符串 "This is a test"。

(2) 调用 fflush 函数将缓冲区的内容写入文件。

(3) 不关闭该文件, 以读的方式打开文件 test.txt, 并用指针 stream2 指向该文件, 试图将刚刚写入的字符串读入到内存缓冲区中。如果读入成功, 打印出该字符串, 否则报错。

实践证明, 例程 10-6 的输出结果是在屏幕上显示字符串 "This is a test"。

两例程运行结果不同的原因在于: 例程 10-5 中将文件打开后, 指针 stream1 指向的是该文件的内存缓冲区, 将字符串写入后也只是写到了文件的内存缓冲区中, 而并没有写到磁盘上的文件中。而当以读的方式打开该文件时, 该文件中的内容实际为空, 也就是 stream2 指向的缓冲区中内容为空, 因此读文件发生错误。而例程 10-6 中, 在写完文件后调用函数 fflush, 将缓冲区的内容写到文件中, 这样再以读的方式打开该文件时, 文件中已经存有了字符串, 因此可以正常读出。



注意：如果在写完文件后调用函数 `fclose` 关闭该文件，同样可以达到将缓冲区的内容写到文件中的目的，但是那样系统开销较大。

## 10.7 fgetc 从流中读取字符函数

函数原型： `int fgetc(FILE *fp);`

头文件： `#include <stdio.h>`

是否是标准函数： 是

函数功能：从流中读取字符，即从 `fp` 所指定的文件中取得下一个字符。这里需要注意，在每读取完一个字符时 `fp` 会自动向下移动一字节，这样程序员在编程时就不用再对 `fp` 控制了。这种功能在许多读写函数中都有体现。

返回值：返回所得到的字符，若读入错误，返回 `EOF`。

### 例程 10-7 应用 `fgetc` 函数从文件中自动读取字符。

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
    FILE *fp;
    char string[] = "This is a test";
    char ch;
    /* 以读写方式打开一个名为 test.txt 的文件 */
    fp = fopen("test.txt", "w+");
    /* 向文件中写入字符串 string */
    fwrite(string, strlen(string), 1, fp);
    /* 将 fp 指针指向文件首 */
    fseek(fp, 0, SEEK_SET);
    do
    {
        /* 从文件中读一个字符 */
        ch = fgetc(fp);
        /* 显示该字符 */
        putchar(ch);
    } while (ch != EOF);
    fclose(fp);
    return 0;
}
```

#### 例程说明：

- (1) 首先程序以读写方式打开一个名为 `test.txt` 的文件，并向该文件中写入一个字符串。
- (2) 然后应用 `fseek` 函数将文件指针 `fp` 定位在文件的开头，循环地将字符逐一读出。这里每读出一个字符，指针 `fp` 会自动地而后移一字节，直至读到文件尾（`EOF` 标志）循环才停止。因为 `fgetc` 函数的返回值为得到的字符，所以用一个字符型变量 `ch` 来接受读出的字符。
- (3) 最后在屏幕上显示运行结果： `This is a test.`



## 10.8 fgetpos 取得当前文件的句柄函数

函数原型: `int fgetpos( FILE *stream, fpos_t *pos );`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 取得当前文件的指针所指的位置, 并把该指针所指的位置数存放到 `pos` 所指的对象中。`pos` 值以内部格式存储, 仅由 `fgetpos` 和 `fsetpos` 使用。其中 `fsetpos` 的功能与 `fgetpos` 相反, 将在后节给予说明。

返回值: 成功返回 0, 失败返回非 0, 并设置 `errno`。

### 例程 10-8 应用 fgetpos 函数取得当前文件的指针所指的位置。

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    FILE *fp;
    char string[] = "This is a test";
    fpos_t pos;
    /*以读写方式打开一个名为 test.txt 的文件*/
    fp = fopen("test.txt", "w+");
    /*将字符串写入文件*/
    fwrite(string, strlen(string), 1, fp);
    /*取得指针位置并存入&pos 所指向的对象*/
    fgetpos(fp, &pos);
    printf("The file pointer is at byte %ld\n", pos);
    /*重设文件指针的位置*/
    fseek(fp, 3, 0);
    /*再次取得指针位置并存入&pos 所指向的对象*/
    fgetpos(fp, &pos);
    printf("The file pointer is at byte %ld\n", pos);
    fclose(fp);
    return 0;
}
```

#### 例程说明:

(1) 首先程序以读写方式打开一个名为 `test.txt` 的文件, 并把字符串 "This is a test" 写入文件。注意: 字符串共 14 字节, 地址为 0~13。用 `fwrite` 函数写入后, 文件指针自动指向文件最后一字节的下一个位置, 即这时的 `fp` 的值应该是 14。

(2) 用 `fgetpos` 函数取得指针位置并存入 `&pos` 所指向的对象, 此时, `pos` 中的内容为 14, 然后在屏幕上显示 `The file pointer is at byte 14`。

(3) 用 `fseek` 函数重设文件指针的位置, 让 `fp` 的值为 3, 即指向文件中第 4 字节。再次取得指针位置并存入 `&pos` 所指向的对象, 然后在屏幕上显示 `The file pointer is at byte 3`。

## 10.9 fgets 从流中读取字符串函数

函数原型: `char *fgets(char *string, int n, FILE *fp);`



头文件: #include<stdio.h>

是否是标准函数: 是

函数功能: 从 fp 所指的文件中读取一个长度为 (n-1) 的字符串, 并将该字符串存入以 string 为起始地址的缓冲区中。fgets 函数有三个参数, 其中 string 为缓冲区首地址, n 规定了要读取的最大长度, fp 为文件指针。

返回值: 返回地址 string, 若遇到文件结束符或出错, 返回 NULL。用 feof 或 ferror 判断是否出错。

#### 例程 10-9 用 fgets 函数从文件中读取字符串。

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    FILE *fp;
    char string[] = "This is a test";
    char str[20];
    /* 以读写的方式打开一个名为 test.txt 的文件 */
    fp = fopen("test.txt", "w+");
    /* 将字符串写入文件 */
    fwrite(string, strlen(string), 1, fp);
    /* 文件指针定位在文件开头 */
    fseek(fp, 0, SEEK_SET);
    /* 从文件中读一个长为 strlen(string) 的字符串 */
    fgets(str, strlen(string)+1, fp);
    /* 显示该字符串 */
    printf("%s", str);
    fclose(fp);
    return 0;
}
```

#### 例程说明:

(1) 首先以读写的方式打开一个名为 test.txt 的文件, 并将字符串写入文件。应用 fseek 函数将文件指针定位在文件开头。

(2) 从文件中读取一个长为 strlen(string) 的字符串, 把字符串读到以 str 为首地址的数组中。这里应注意第二个参数若为 n, 则表示从 fp 所指的文件中读取一个长度为 (n-1) 的字符串。因此, 这里的参数为 strlen(string)+1, 表示读取一个长度为 strlen(string) 的字符串。

(3) 最后在屏幕上显示该字符串。

## 10.10 fprintf 格式化输出函数

函数原型: int fprintf(FILE \*fp, char \*format[, argument,...]);

头文件: #include<stdio.h>

是否是标准函数: 是

函数功能: 把 argument 的值以 format 所指定的格式输出到 fp 指向的文件中。函数的理解和 printf 类似, 在一般的使用中, 第二个参数可以是一个字符串的头指针, 也可以是一个字符串。例如: fprintf(fp, "Cannot open this file!!")意思是把字符串 Cannot open this file!!



输出到文件 fp 中去。该函数一般用作终端的出错提示或是在磁盘中生成错误报告。

返回值：如果正确返回实际输出字符数，若错误则返回一个负数。

例程 10-10 用 fprintf 函数向终端发出出错提示。

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    /*以只读方式打开名为 test.txt 的文件*/
    if ((fp = fopen("\\test.txt", "rt"))
        == NULL)
    {
        fprintf(stderr, "Cannot open this file!!\n");
        return 1;    /*若该文件不能打开，在屏幕上显示出错提示*/
    }
    /*若该文件能够打开，在屏幕上显示正确提示*/
    fprintf(stderr, "Have open this file!!\n");
    return 0;
}
```

例程说明：

- (1) 首先以只读方式打开名为 test.txt 的文件，如果文件不能打开，则返回 NULL。
- (2) 若该文件不能打开，在屏幕上显示出错提示。
- (3) 若该文件能够打开，在屏幕上显示正确提示。

注意：该函数中第一个参数是 stderr，这是 C 语言中标准出错输出指针，它指向标准的出错输出文件，也就是显示器。因为在操作系统中，I/O 设备都是用文件进行管理的，因此设备都配有相应的控制文件。在 C 语言中，有三个文件与终端联系，因此系统定义了三个文件指针，如表 10-2 所示。在系统运行时，程序自动打开这三个标准文件。

表 10-2 标准文件指针

设备文件	文件指针
标准输入	stdin
标准输出	stdout
标准出错输出	stderr

本例程的运行结果如下。

- (1) 如果不能打开文件：

```
Cannot open this file!!
```

- (2) 如果可以打开文件：

```
Have open this file!!
```

例程 10-11 用 fprintf 函数在磁盘中生成错误报告。

```
#include <stdio.h>
int main(void)
{
    FILE *fp1, *fp2;
    /*以只读方式打开名为 test.txt 的文件*/
```



```
if ((fp1 = fopen("text.txt", "rt"))
    == NULL)
{
    /*若文件打不开，则生成错误报告*/
    fp2=fopen("report.txt", "w");
    fprintf(fp2, "Cannot open this file!!\n");
    return 1;
}
return 0;
}
```

#### 例程说明：

- (1) 首先以只读方式打开名为 test.txt 的文件，如果文件不能打开，则返回 NULL。
- (2) 若该文件不能打开，则以写的方式打开一个名为 report.txt 的文件，并按照格式要求向文件中写入字符串 "Cannot open this file!!\n"，即生成了一个错误报告。

注意：这里函数 fprintf 的第一个参数为文件指针，是用户自定义的，与例程 10-10 的系统定义的文件指针 stderr 不同。fprintf 函数的使用与 printf 函数类似，其实 printf 函数是 fprintf 函数的一个特例。printf 函数只能向标准输出文件（显示器）输出数据，而 fprintf 函数也可以向一般用户定义的文件输出数据。

## 10.11 fputc 向流中输出字符函数

函数原型：int fputc(char ch, FILE \*fp);

头文件：#include <stdio.h>

是否是标准函数：是

函数功能：将字符 ch 输出到 fp 指向的文件中。该函数与前边提到的 fgetc 是相对的，第一个参数 ch 是字符型变量，函数将该变量中的字符输出到 fp 指向的文件中。

返回值：成功返回该字符，否则返回非 0。

#### 例程 10-12 应用 fputc 向文件输出字符。

```
#include <stdio.h>
int main(void)
{
    /*初始化字符数组 str */
    char str[] = "This is a test";
    int i = 0;
    /*将数组中的字符循环输出至屏幕*/
    while (str[i])
    {
        fputc(str[i], stdout);
        i++;
    }
    return 0;
}
```

#### 例程说明：

- (1) 首先初始化字符数组 str。在 C 语言中初始化数组时，系统会自动在数组最后添加 '\0'，以表示该字符串结束。



(2) 将数组中的字符循环输出至屏幕。这里注意两点:

- ✧ 该循环以'\0'作为结束标志,即循环碰到'\0'时结束。
- ✧ 函数 fputc 的第二个参数是 stdout,代表标准输出文件指针,在屏幕上显示该字符串。

本例程的运行结果为:

```
This is a test
```

## 10.12 fputs 向流中输出字符串函数

函数原型: `int fputs(char *string, FILE *fp);`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 将 string 所指的字符串输出到 fp 指向的文件中。该函数与 fgets 相对,第一个参数为字符串指针,不同的是 fputs 函数没有字符串长度的限制,只是将 string 指向的字符串输出到文件中。

返回值: 成功返回 0, 否则返回非 0。

### 例程 10-13 应用 fputs 函数向文件中输出字符串。

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    char str[]="This is a test!";
    /*以写的方式打开名为 test.txt 的文件*/
    fp=fopen("test.txt","w");
    /*将字符串写入文件*/
    fputs(str,fp);
    fclose(fp);
    return 0;
}
```

例程说明:

- (1) 首先用字符数组 str 存储一个字符串,并以写的方式打开名为 test.txt 的文件。
- (2) 再用 fputs 函数将该字符串输出到 test.txt 的文件中。

## 10.13 fread 从流中读取字符串函数

函数原型: `int fread(void *buf, int size, int count, FILE *fp);`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 从 fp 指向的文件中读取长度为 size 的 count 个数据项,并将它输入到以 buf 为首地址的缓冲区中。此时,文件指针 fp 会自动增加实际读入数据的字节数,即 fp 指向最后读入字符的下一个字符位置。



**返回值：**返回实际读入数据项的个数，即 count 值。若遇到错误或文件结束则返回 0。

#### 例程 10-14 应用 fread 函数从文件中读取数据到缓冲区。

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    FILE *fp;
    char str[] = "this is a test";
    char buf[20];

    if ((fp = fopen("test.txt", "w+"))
        == NULL)
    {
        fprintf(stderr,
            "Cannot open output file!!\n");
        return 1;
    }

    /* 向文件中写入字符串数组中的数据 */
    fwrite(str, strlen(str), 1, fp);
    /* 将文件指针定位到文件开头 */
    fseek(fp, SEEK_SET, 0);
    /* 把文件中的数据读出并显示 */
    fread(buf, strlen(str), 1, fp);
    printf("%s\n", buf);
    fclose(fp);
    return 0;
}
```

#### 例程说明：

- (1) 程序首先以读写方式打开名为 test.txt 的文件。这里有一个判断，若打开文件失败，则在屏幕上显示出错信息。
- (2) 应用 fwrite 函数向文件写入数据。有关 fwrite 函数后面做详细介绍。
- (3) 应用 fseek 函数将文件指针定位到文件开头。
- (4) 应用 fread 函数把文件中的数据读入内存。这里读取一个长度为 strlen(str) 的字符串，并将该字符串存入以 buf 为首地址的内存缓冲区中。
- (5) 显示该字符串。

## 10.14 freopen 替换文件中数据流函数

**函数原型：**FILE \*freopen(char \*filename, char \*type, FILE \*fp);

**头文件：**#include<stdio.h>

**是否是标准函数：**是

**函数功能：**关闭 fp 所指向的文件，并将该文件中的数据流替换到 filename 所指向的文件中去。该函数共有三个参数：filename 是文件流将要替换到的文件名路径；type 是文件打开的方式，它与 fopen 中的文件打开方式类似；fp 是要被替换的文件指针。

**返回值：**返回一个指向新文件的指针，即指向 filename 文件的指针。若出错，则返回 NULL。



**例程 10-15** 关闭一个终端，并将数据流替换至一个新文件中。

```
#include <stdio.h>
int main(void)
{
    FILE *Nfp;
    /*替换标准输出文件上的数据流到新文件 test.txt*/
    if (Nfp=freopen("test.txt", "w", stdout)
        == NULL)
        fprintf(stderr, "error redirecting stdout\n");
    /*标准输出文件上的数据流将会被替换到新文件中*/
    printf("This will go into a file.");

    /*关闭标准输出文件*/
    fclose(stdout);
    /*关闭新生成的文件*/
    fclose(Nfp);
    return 0;
}
```

**例程说明：**

(1) 首先程序以写方式打开名为 test.txt 的文件，将标准输出文件上的数据流 "This will go into a file." 替换到新生成的文件 test.txt 中。freopen 函数返回一个指向新文件的指针，即指向文件 test.txt 的指针，并将它存放到 Nfp 中。

(2) 然后关闭标准输出文件 fclose(stdout)。

(3) 最后关闭新生成的文件 fclose(Nfp)。

(4) 本程序的执行结果是在当前目录下生成一个 test.txt 文件，并将原终端的数据流 "This will go into a file." 重新写入 test.txt 文件中。

## 10.15 fscanf 格式化输入函数

**函数原型：** int fscanf(FILE \*fp, char \*format[,argument...]);

**头文件：** #include<stdio.h>

**是否是标准函数：** 是

**函数功能：** 从 fp 所指向的文件中按 format 给定的格式读取数据到 argument 所指向的内存单元。其中 argument 是指针，指针类型要与输入的格式 format 一致。例如：fscanf(stdin, "%d", &i)，其中 &i 是整型的指针，输入的格式 format 也要为整型，即 "%d"。

**返回值：** 返回已输入的数据个数。若错误或文件结束则返回 EOF。

**例程 10-16** 应用 fscanf 函数从终端向内存输入数据。

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    char ch;
    printf("Please input an character ");
    /* 从标准输入文件中读取一个字符，并送至 ch */
    if (fscanf(stdin, "%c", &ch))
        printf("The character was: %c\n",ch);
    else
```



```
{
    /*出错提示*/
    fprintf(stderr, "Error reading an character from stdin!!\n");
    exit(1);
}
return 0;
}
```

例程说明:

- (1) 首先程序提示用户从键盘输入一个字符。这时，标准输入文件指针 `stdin` 指向该字符。
- (2) 调用 `fscanf` 函数从标准输入文件中读取一个字符，并送至 `ch`。这里应该注意两点：
  - ✧ `fscanf` 函数的第一个参数不是用户定义的文件指针，而是系统定义的标准输入文件指针 `stdin`，但用法与用户定义的文件指针类似。
  - ✧ `&ch` 是指向字符型数据的指针，因此，输入的格式 `format` 也要为字符型 “`%c`”，它们必须保持一致。
- (3) 该函数的使用与 `scanf` 类似。`scanf` 函数是 `fscanf` 函数的一个特例，它只能从标准输入文件（键盘终端）中输入数据，而 `fscanf` 函数则可以从一般用户定义的文件中输入数据。

10.16 fseek 文件指针定位函数

函数原型: `int fseek(FILE *fp, long offset, int base);`

头文件: `#include<stdio.h>`

是否是标准函数: 是

函数功能: 重定位流上的文件指针，即将 `fp` 指向的文件位置指针移向以 `base` 为基准、以 `offset` 为偏移量的位置。该函数有三个参数: `fp` 为文件指针; `offset` 为偏移量，即位移 `offset` 字节; `base` 为指针移动的基准，即起始点。其中，基准 `base` 用 0、1 或 2 表示。在 ANSI C 标准指定的名字如表 10-3 所示。

表 10-3 ANSI C标准指定的起始点名字

起始点	名字	数字代码
文件当前位置	SEEK_CUR	1
文件开始位置	SEEK_SET	0
文件末尾位置	SEEK_END	2

偏移量用长整型数表示。ANSI C 标准规定，在数的末尾加 `L` 就表示长整型数。该函数在随机读取较长的顺序文件时是很有用的。

返回值: 成功返回 0，否则返回非 0。

例程 10-17 文件指针的定位演示。

```
#include <stdio.h>
void main( void )
```



```

{
    FILE *fp;
    char line[81];
    int result;
    /*以读写的方式打开名为 test.txt 的文件*/
    fp = fopen( "test.txt", "w+" );
    if( fp == NULL )
        printf( "The file test.txt was not opened! \n" );
    else
    {
        /*按照规定格式将字符串写入文件*/
        fprintf( fp, "The fseek begins here: "
                "This is the file 'test.txt'.\n" );
        /*将文件指针定位到离文件头 23 字节处*/
        result = fseek( fp, 23L, SEEK_SET);
        if( result )
            perror( "Fseek failed" );
        else
        {
            printf( "File pointer is set to middle of first line.\n" );
            /*从 fp 指向的文件中读取字符串*/
            fgets( line, 80, fp );
            /*显示读取的字符串*/
            printf( "%s", line );
        }
        fclose(fp);
    }
}

```

#### 例程说明:

- (1) 首先程序以读写方式打开名为 test.txt 的文件。
- (2) 然后应用 fprintf 函数按照规定格式将字符串"The fseek begins here: "、"This is the file 'test.txt'.\\n"写入文件。
- (3) 再将文件指针定位到离文件头 23 字节处,即将文件指针 fp 定位在字符串"This is the file 'test.txt'.\\n"的开头。
- (4) 最后应用 fgets 函数从 fp 指向的文件中读取字符串,并显示在屏幕上。

本程序的运行结果为:

```

File pointer is set to middle of first line.
This is the file 'test.txt'.

```

## 10.17 fsetpos 定位流上的文件指针函数

函数原型: int fsetpos(FILE \*fp, const fpos\_t \*pos);

头文件: #include<stdio.h>

是否是标准函数: 是

函数功能: 将文件指针定位在 pos 指定的位置上。该函数的功能与前面提到的 fgetpos 相反,是将文件指针 fp 按照 pos 指定的位置在文件中定位。pos 值以内部格式存储,仅由 fgetpos 和 fsetpos 使用。

返回值: 成功返回 0, 否则返回非 0。



**例程 10-18** 应用 fsetpos 函数定位文件指针。

```
#include <stdio.h>
void main( void )
{
    FILE *fp;
    fpos_t pos;
    char buffer[50];
    /*以只读方式打开名为 test.txt 的文件*/
    if( (fp = fopen( "test.txt", "rb" )) == NULL )
        printf( "Trouble opening file\n" );
    else
    {
        /*设置 pos 值*/
        pos = 10;
        /*应用 fsetpos 函数将文件指针 fp 按照 pos 指定的位置在文件中定位*/
        if( fsetpos( fp, &pos ) != 0 )
            perror( "fsetpos error" );
        else
        {
            /*从新定位的文件指针开始读取 16 个字符到 buffer 缓冲区*/
            fread( buffer, sizeof( char ), 16, fp );
            /*显示结果*/
            printf( "16 bytes at byte %ld: %.16s\n", pos, buffer );
        }
    }
    fclose( fp );
}
```

**例程说明：**

- (1) 首先程序以只读方式打开名为 test.txt 的文件。在这里，test.txt 文件中已存入字符串"This is a test for testing the function of fsetpos"。
- (2) 将 pos 设置为 10，应用 fsetpos 函数将文件指针 fp 按照 pos 指定的位置在文件中定位。这样，文件指针 fp 指向字符串中 test 的字母 t。
- (3) 再从新定位的文件指针开始读取 16 个字符到 buffer 缓冲区，也就是读取字符串"test for testing"到缓冲区 buffer。
- (4) 最后显示结果：16 bytes at byte 10: test for testing。

## 10.18 ftell 返回当前文件指针位置函数

函数原型：long ftell(FILE \*fp);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：返回当前文件指针的位置。这个位置是指当前文件指针相对于文件开头的位移量。

返回值：返回文件指针的位置，若出错则返回-1。

**例程 10-19** 应用 ftell 返回文件指针位置。

```
#include <stdio.h>
```



```
int main(void)
{
    FILE *fp;
    fp = fopen("test.txt", "w+");
    /*按照格式要求将字符串写入文件*/
    fprintf(fp, "This is a test");
    /*读出文件指针 fp 的位置*/
    printf("The file pointer is at byte %ld\n", ftell(fp));
    fclose(fp);
    return 0;
}
```

### 例程说明:

(1) 首先以写方式打开名为 test.txt 的文件, 按照格式要求将字符串写入文件。注意: 字符串共 14 个字符, 地址为 0~13。调用 fprintf 函数后, 文件指针自动移到读入的最后一个字符的下一个位置, 本例中就是文件的结束符, 它的地址是 14。

(2) 应用 ftell 函数读出文件指针 fp 的位置。

注意: 本题中 ftell 函数的返回值实际上就是该文件的长度。在实际的应用中, 函数 ftell 常用来计算文件的长度。

## 10.19 fwrite 向文件写入数据函数

函数原型: int fwrite(void \*buf, int size, int count, FILE \*fp);

头文件: #include<stdio.h>

是否是标准函数: 是

函数功能: 将 buf 所指向的 count\*size 字节的数据输出到文件指针 fp 所指向的文件中。该函数与 fread 相对, 输出数据后, 文件指针 fp 自动指向输出的最后一个字符的下一个位置。该函数常用于将数据块分批输出到文件中。

返回值: 返回实际写入文件数据项个数。

### 例程 10-20 应用 fwrite 函数向文件中写入数据块。

```
#include <stdio.h>
struct exp_struct
{
    int i;
    char ch;
};
int main(void)
{
    FILE *fp;
    struct exp_struct s;
    /*以写的方式打开名为 test.txt 的文件*/
    if ((fp = fopen("test.txt", "wb")) == NULL)
    {
        fprintf(stderr, "Cannot open the test.txt");
        return 1;
    }
    /*向结构体中的成员赋值*/
    s.i = 0;
```



```
s.ch = 'A';
/* 将一个结构体数据块写入文件 */
fwrite(&s, sizeof(s), 1, fp);
fclose(fp);
return 0;
}
```

#### 例程说明:

- (1) 程序先声明一个结构体类型 `struct exp_struct`, 这样一个结构体变量就是一个小数数据块。
- (2) 再以写方式打开名为 `test.txt` 的文件。
- (3) 然后向结构体中的成员变量赋值, 并将赋值好的数据块应用 `fwrite` 函数写入 `fp` 所指向的文件中。这里参数 `sizeof(s)` 是该结构体变量的大小, 1 指只写入文件 1 个数据块。
- (4) 最后关闭该文件。

## 10.20 getc 从流中读取字符函数

函数原型: `int getc(FILE *fp);`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 从 `fp` 所指向的文件中读取一个字符。该函数与前面所讲到的 `fgetc` 作用类似。读取字符后, 文件指针 `fp` 自动指向下一个字符。

返回值: 返回所读的字符, 若文件结束或出错则返回 `EOF`。

### 例程 10-21 应用 getc 函数从标准输入文件中读取一个字符。

```
#include <stdio.h>
int main(void)
{
    char ch;
    printf("Input a character:");
    /* 从标准输入文件中读取一个字符 */
    ch = getc(stdin);
    /* 显示该字符 */
    printf("The character input was: '%c'\n", ch);
    return 0;
}
```

#### 例程说明:

本程序是从标准输入文件（键盘）中读取一个字符, 存入变量 `ch`, 并显示在屏幕上。

### 例程 10-22 应用 getc 函数从一般文件中读取字符。

```
#include <stdio.h>
void main( void )
{
    FILE *fp;
    char ch;
    /* 以写的方式打开名为 test.txt 的文件 */
    fp=fopen("test.txt", "w");
    /* 写入字符串 */
}
```



```

fprintf(fp, "This is a test.");
fclose(fp);
/*再以读的方式打开名为 test.txt 的文件*/
fp=fopen("test.txt", "r");
/*将文件指针指向文件开头*/
fseek(fp, 0L, SEEK_SET);
/*应用 getc 函数从文件中循环读取字符并显示出来*/
while(feof(fp) == 0)
{
    ch=getc(fp);
    printf("%c", ch);
}
fclose(fp);
return 0;
}

```

**例程说明:**

- (1) 首先以写方式打开名为 test.txt 的文件，并将字符串"This is a test."写入文件。
- (2) 再以读方式打开名为 test.txt 的文件，并将文件指针指向文件开头。
- (3) 最后应用 getc 函数从文件中循环读取字符，直到文件结束为止，并将读取的字符显示到终端屏幕。

---

注意：本例程与例程 10-20 不同，例程 10-20 是从标准输入文件（键盘）中读取一个字符，本例是从一般文件中读取字符，函数的参数不同。

---

## 10.21 getchar 从标准输入文件中读取字符函数

函数原型：int getchar(void);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：从标准输入文件 stdin（键盘）中读取一个字符。该函数与 getc 类似，但参数为空，它只能从标准输入文件中读取字符，而不能读取用户自定义的文件。getchar 函数在编程时多用于接收回车、换行符。

返回值：返回所读的字符，若文件结束或出错则返回-1。

### 例程 10-23 应用 getchar 函数从标准输入设备读取下一个字符。

```

#include <stdio.h>
int main(void)
{
    int c;
    /*从键盘上接收字符并显示，直到键入换行符为止*/
    while ((c = getchar()) != '\n')
        printf("%c", c);
    return 0;
}

```

**例程说明:**

程序从键盘上接收字符并显示，当接收到换行符'\n'时，程序结束。

## 10.22 gets 从标准输入文件中读取字符串函数

函数原型: `char *gets(char *buf);`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 从标准输入文件 `stdin` (键盘) 中读取一个字符串, 并把该字符串存入以 `buf` 为首地址的缓冲区中。该函数与 `fgets` 类似, 但它只能从标准输入文件 `stdin` 中读取字符串, 而且没有长度限制。

返回值: 返回其参数, 即缓冲区指针 `buf`, 若出错则返回 `NULL`。

### 例程 10-24 应用 gets 函数从键盘读取字符串。

```
#include <stdio.h>
int main(void)
{
    char string[30];
    printf("Input a string:");
    /*从终端输入字符串, 注意不要超过 30 个字符*/
    gets(string);
    /*显示该字符串*/
    printf("The string input was: %s\n", string);
    return 0;
}
```

例程说明:

- (1) 首先程序开辟一个可容纳 30 个字符的字符串数组空间, 并在屏幕上提示用户输入一个字符串。
- (2) 应用 `gets` 函数接收键盘输入的字符串, 并把它存储到以 `string` 为首地址的缓冲区中。
- (3) 最后将以 `string` 为首地址的缓冲区中的内容显示出来, 即在屏幕上显示输入的字符串。

## 10.23 perror 打印系统错误信息函数

函数原型: `void perror(char *string);`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 将错误信息输出到标准出错输出 `stderr`。该函数的参数是字符串指针, 指向错误信息字符串。也可以是一个字符串, 直接在参数中输入要显示的错误信息。但要注意, 完整的错误信息不仅包括用户在参数中自己定义的字符串, 还包括一个冒号、系统报错信息和一个换行符。该函数主要用作向终端进行错误提示。

返回值: 无。



**例程 10-25** 应用 perror 函数显示错误信息。

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    /*企图以读的方式打开文件 test.txt*/
    fp = fopen("test.txt", "r");
    if (fp==NULL)
        /*该文件不存在，在终端显示错误信息*/
        perror("Unable to open file for reading");
    return 0;
}
```

**例程说明：**

- (1) 首先程序企图以读方式打开文件 test.txt，但该文件在这里并不存在。
- (2) 然后利用函数 perror 在终端显示错误信息"Unable to open file for reading: No such file or directory"。

具体的运行结果如下：

```
Unable to open file for reading: No such file or directory
```

注意：完整的错误信息包括用户自定义字符串、冒号、系统报错信息和换行符。

## 10.24 printf 产生格式化输出的函数

**函数原型：**int printf( const char \*format [, argument]... );

**头文件：**#include<stdio.h>

**是否是标准函数：**是

**函数功能：**按 format 指向的格式字符串所规定的格式，将输出表列 argument 的值输出到标准输出设备。该函数与 fprintf 类似，但只能将 argument 的值输出到标准输出设备 stdout，即显示器屏幕，而不能输出到用户自定义的文件中。

**返回值：**输出字符的个数，若出错则返回一个负数。

**例程 10-26** 应用 printf 函数输出字符串。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    int a=1;
    char ch='r';
    char str[]="This is a test!";
    printf("Output a string.\n");
    printf("%s",str);
    printf("The integer is %d\n",a);
    printf("The character is %c\n",ch);
    return 0;
}
```

例程说明:

- (1) 首先在标准输出设备 stdout, 即显示器屏幕上打印出 Output a string.。
- (2) 再打印出字符串 str 中的内容: This is a test!。
- (3) 然后打印出整型数 a: The integer is 1。
- (4) 再打印出字符 ch: The character is r。

## 10.25 putc 向指定流中输出字符函数

函数原型: `int putc(int ch, FILE *fp);`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 向指定的文件输出一个字符。该函数有两个参数, ch 是用户指定的字符, fp 是文件指针。函数将字符 ch 输出到 fp 指定的文件中。

返回值: 返回输出的字符 ch, 若出错则返回 EOF。

### 例程 10-27 应用 putc 函数向标准输出文件输出字符。

```
#include <stdio.h>
int main(void)
{
    char str[] = "Hello world!";
    int i = 0;
    while (str[i]){
        putc(str[i], stdout);
        i++;
    }
    return 0;
}
```

例程说明:

- (1) 首先将字符串 "Hello world!" 存入字符串数组 str 中。
- (2) 循环地将数组 str 中的内容输出到标准输出文件 (显示器) 上。

### 例程 10-28 应用 putc 函数向用户自定义文件输出字符。

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    int i = 0;
    char str[] = "This is a test!";
    fp = fopen("test.txt", "w");
    while (str[i]){
        putc(str[i], fp);
        i++;
    }
    fclose(fp);
    return 0;
}
```



例程说明：

- (1) 首先将字符串 "This is a test!" 存入字符串数组 str 中。
- (2) 以写方式打开一个名为 test.txt 的文件，并利用函数 putc 将数组 str 中的内容输出到文件 test.txt 中。
- (3) 关闭文件。

注意：putc 函数既可以向标准输出文件输出字符，又可以向用户自定义文件输出字符。而且，每向文件输出一个字符时，文件指针就会自动向后移一字节。

## 10.26 putchar 向标准输出文件上输出字符

函数原型：int putchar(char ch);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：将字符串 ch 输出到标准输出文件 stdout（显示器）上。

返回值：返回输出的字符 ch，若出错则返回 EOF。

**例程 10-29** 应用 putchar 函数在屏幕上显示字符。

```
#include <stdio.h>
int main()
{
    char str[]="This is a test!\n";
    int i=0;
    while(str[i]){
        putchar(str[i]);
        i++;
    }
}
```

例程说明：

- (1) 首先将字符串 "This is a test!\n" 存入字符串数组 str 中。
- (2) 应用 putchar 函数，循环地将字符串输出到标准输出文件（终端屏幕）上。

注意：putchar 函数与 putc 函数不同，它只能向标准输出文件，也就是向终端屏幕上输出字符。而 putc 函数既可以向标准输出文件上输出字符，又可以向一般用户自定义文件上输出字符。

## 10.27 puts 将字符串输出到终端函数

函数原型：int puts(char \*string);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：将 string 指向的字符串输出到标准输出设备（stdout），并将 '\0' 转换为回车换行符 '\n'。在 C 语言中，字符串以 '\0' 结尾。该函数将字符串输出到标准输出设备的同时，

将字符串的结束标志转换为回车换行。

返回值：成功则返回换行符，若失败则返回 EOF。

### 例程 10-30 应用 puts 函数向终端输出字符串。

```
#include <stdio.h>
int main(void)
{
    char string[] = "This is a test!\n";
    puts(string);
    return 0;
}
```

例程说明：

- (1) 首先将字符串 "This is a test!\n" 存入以 string 为首地址的缓冲区。
- (2) 应用 puts 函数将该字符串显示在标准输出设备上。

注意：puts 函数将字符串的结尾标志 '\0' 转换为回车换行符 '\n'。因此，程序运行的结果为：

```
This is a test!
```

## 10.28 remove 删除文件函数

函数原型：int remove(char \*filename);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：删除以 filename 为文件名的文件。该函数的参数为欲删除文件的文件名，如果是单纯的文件名，就表明删除当前文件夹下的文件，否则要写明文件的路径。

返回值：成功删除文件返回 0，否则返回-1。

### 例程 10-31 应用 remove 函数删除文件。

```
#include <stdio.h>
void main()
{
    if( remove( "test.txt" ) == -1 )
        perror( "Could not delete test.txt!!" );
    else
        printf( "Deleted test.txt \n" );
}
```

例程说明：

- (1) 首先要在当前文件夹下建立文件 test.txt。
- (2) 再利用 remove 函数删除该文件。若删除成功，则在终端显示字符串 "Deleted test.txt"，否则显示字符串 "Could not delete test.txt!!: No such file or directory"。

注意：前面已经讲过 perror 函数是按照一定格式向终端输出错误信息，因此，若删除文件成功，程序运行的结果为：

```
Deleted test.txt
```



若删除失败，则程序运行的结果为：

```
Could not delete test.txt!!!: No such file or directory
```

这里，利用 `perror` 函数显示的完整的错误信息包括：用户自定义字符串、冒号、系统报错信息、换行符。

## 10.29 rename 重命名文件函数

函数原型：int rename(char \*oldname, char \*newname);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：把 `oldname` 所指的文件名改为 `newname` 所指的文件名。该函数有两个参数，`oldname` 为旧的文件名，`newname` 为欲改成的新文件名。应当注意，`oldname` 所指的文件一定要存在，`newname` 所指的文件一定不存在。应用该函数可将一个文件的旧文件名 `oldname` 改为新文件名 `newname`，但不能改变文件的路径。

返回值：成功返回 0，出错则返回-1。

### 例程 10-32 应用 rename 函数重命名文件。

```
#include <stdio.h>
int main(void)
{
    if( rename("oldname.txt", "newname.txt")==0)
        printf("Rename successful!!");
    else
        printf("Rename fail!!");
}
```

例程说明：

(1) 首先在当前文件夹下建立一个文件 `oldname.txt`。

(2) 应用 `rename` 函数重命名该文件，将其改名为 `newname.txt`。若重命名成功，在屏幕上显示 "Rename successful!!" 提示字符串；否则显示 "Rename fail!!" 提示字符串。

## 10.30 rewind 重置文件指针函数

函数原型：void rewind(FILE \*stream);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：将文件指针 `fp` 重新定位在文件开头，并清除文件的结束标志和错误标志。该函数与函数 `fseek` 功能类似，但不同的是，该函数可以清除文件的结束标志和错误标志，而函数 `fseek` 不能；另外，该函数不能像函数 `fseek` 一样返回一个值表明操作是否成功，因为该函数无返回值。

返回值：无。



**例程 10-33 应用函数 rewind 重定位文件指针。**

```
#include <stdio.h>
void main( void )
{
    FILE *fp;
    int data1, data2;
    data1 = 1;
    data2 = 2;
    if( (fp = fopen( "test.txt", "w+" )) != NULL )
    {
        fprintf( fp, "%d %d", data1, data2 );
        printf( "The values written are: %d and %d\n", data1, data2 );
        data1=0;
        data2=0;
        rewind( fp );
        fscanf( fp, "%d %d", &data1, &data2 );
        printf( "The values read are: %d and %d\n", data1, data2 );
        fclose( fp );
    }
}
```

**例程说明:**

- (1) 程序首先以写方式打开一个名为 test.txt 的文件。
- (2) 应用 fprintf 函数向该文件写入 data1, data2 两个整型数, 其值分别为 1, 2。此时, 文件指针 fp 已指向文件尾。
- (3) 在屏幕上显示这两个数。
- (4) 再将变量 data1 和 data2 置 0。
- (4) 应用 rewind 函数重定位文件指针, 将文件指针 fp 重新定位在文件开头。
- (5) 再应用 fscanf 函数向 data1, data2 两个变量中读入文件中的数字。
- (6) 在屏幕上显示这两个数。

注意: 在应用 fprintf 函数向该文件写入 data1, data2 两个整型数后, 文件指针 fp 会自动指向文件尾。只有再应用函数 rewind、fseek 才能将文件指针重新定位到文件开头, 以便读取文件。本例中, 将 data1 和 data2 置 0 的目的是为了说明应用 fscanf 函数向 data1, data2 两个变量中读入文件中的数字的结果是正确的。

本程序的运行结果为:

```
The values written are: 1 and 2
The values read are: 1 and 2
```

## 10.31 scanf 格式化输入函数

函数原型: int scanf(char \*format[,argument,...]);

头文件: #include<stdio.h>

是否是标准函数: 是

函数功能: 从标准输入设备(键盘)按照 format 指定的格式字符串所规定的格式, 将数据输入到 argument 所指定的内存单元。scanf 函数与 printf 函数相对, 前者是从标准输入设备输入数值, 后者是从标准输出设备输出数值。



返回值：成功返回输入的字符个数，否则遇到结束符返回 EOF，出错则返回 0。

例程 10-34 应用 scanf 函数输入数据。

```
#include <stdio.h>
void main( void )
{
    int i;
    char ch;
    float f;
    printf("Please input an integer:\n");
    scanf("%d",&i);
    getchar();
    printf("Please input a character:\n");
    scanf("%c",&ch);
    getchar();
    printf("Please input an float:\n");
    scanf("%f",&f);
    getchar();
    printf("These values are:%d,%c,%f",i,ch,f);
}
```

例程说明：

- (1)应用 scanf 函数向预先声明的三个变量空间输入一个整型数、一个字符和一个浮点数。
- (2) 在屏幕上显示这三个值。

注意: scanf 函数与 fscanf 函数类似,但只能从标准输入设备文件读取数值,而不能像 fscanf 函数一样从一般用户自定义文件中读取数值。scanf 函数常用作程序设计中数据的输入函数。

10.32 setbuf、setvbuf 指定文件流的缓冲区函数

函数原型: void setbuf(FILE \*fp, char \*buf);  
void setvbuf(FILE \*fp, char \*buf, int type, unsigned size);

头文件: #include<stdio.h>

是否是标准函数: 是

函数功能: 这两个函数使得打开文件后,用户可以建立自己的文件缓冲区,而不必使用 fopen 函数打开文件时设定的默认缓冲区。

setbuf 函数的定义中,参数 buf 指定的缓冲区大小由 stdio.h 中定义的宏 BUFSIZE 的值决定,默认值 default 为 512 字节。而当 buf 为 NULL 时, setbuf 函数将使文件 I/O 不带缓冲区。setvbuf 函数则由 malloc 函数来分配缓冲区,参数 size 指明了缓冲区的长度(必须大于 0),而参数 type 则表明了缓冲的类型,其取值如表 10-4 所示。

表 10-4 setvbuf函数中参数type的取值含义

type 值	含义
_IOFBF	文件全部缓冲,即缓冲区装满后,才能对文件读写
_IOLBF	文件行缓冲,即缓冲区接收到一个换行符时,才能对文件读写
_IONBF	文件不缓冲,此时忽略buf, size的值,直接读写文件,不再经过文件缓冲区缓冲



返回值：无。

### 例程 10-35 应用 setbuf 函数指定文件的缓冲区。

```
#include <stdio.h>
void main( void )
{
    char buf[BUFSIZ];
    FILE *fp1, *fp2;
    if( ((fp1 = fopen( "test1.txt", "a" )) != NULL) &&
        ((fp2 = fopen( "test2.txt", "w" )) != NULL) )
    {
        /* 应用 setbuf 函数给文件流 fp1 指定缓冲区 buf */
        setbuf( fp1, buf );
        /*显示缓冲区地址*/
        printf( "fp1 set to user-defined buffer at: %Fp\n", buf );
        /* 文件流 fp2 不指定缓冲区*/
        setbuf( fp2, NULL );
        /*信息提示不分配缓冲区*/
        printf( "fp2 buffering disabled\n" );
        fclose(fp1);
        fclose(fp2);
    }
}
```

#### 例程说明：

- (1) 首先开辟一个大小为 BUFSIZ 的缓冲区，用作指定文件的缓冲区。这里，BUFSIZE 为 stdio.h 中定义的宏，默认值为 512 字节。
- (2) 以追加方式和写方式打开名为 test1.txt 和 test2.txt 的文件。
- (3) 应用 setbuf 函数给文件流 fp1 指定缓冲区 buf，并在屏幕上显示该首地址，其中 buf 为缓冲区的首地址。
- (4) 文件流 fp2 不指定缓冲区，也就是第二个参数设置为 NULL，信息提示不分配缓冲区。
- (5) 关闭两个文件。

注意：使用 setbuf 函数指定文件的缓冲区时，一定要在文件读写之前。一旦用户自己指定了文件的缓冲区，文件的读写就要在用户指定的缓冲区中进行，而不在系统默认指定的缓冲区中进行。函数 setvbuf 的用法与 setbuf 类似，只是它的缓冲区大小可以动态分配，由函数的参数指定，而且缓冲区的类型也可以由参数指定。

## 10.33 sprintf 向字符串写入格式化数据函数

函数原型：int sprintf(char \*string, char \*format [,argument,...]);

头文件：#include<stdio.h>

是否是标准函数：是

函数功能：将格式化的数据存储在以 string 为首地址的缓冲区中。参数 argument 要被转化为 format 规定的格式，并按照这个规定的格式向字符串数组写入数据。这里应该注意，sprintf 函数与 printf、fprintf 函数不同，前者是向缓冲区（数组）写入格式化数据，后者是



向标准输出文件 (stdout) 和用户自定义文件输出格式化数据。

**返回值:** 返回存储在 string 中数据的字节数。

#### 例程 10-36 应用 sprintf 函数向指定缓冲区写入数据。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    char str[80];
    sprintf(str, "An approximation of Pi is %f\n", M_PI);
    puts(str);
    return 0;
}
```

#### 例程说明:

- (1) 首先开辟一个 80 字节大小的缓冲区 str, 该缓冲区以 str 为首地址。
- (2) 然后将指定的字符串写入缓冲区 str 中。其中, M\_PI 是 math.h 中定义的常量 3.141593。
- (3) 应用 puts 函数向终端输出该字符串。

**注意:** puts 函数的作用是把 str 指定的字符串输出到标准输出设备, 并且将字符串结束标志 '\0' 转换为回车换行符。因此, 该程序运行的结果是:

```
An approximation of Pi is 3.141593
```

除了规定字符串中的换行符外, 程序还将 '\0' 转换为换行符。

## 10.34 sscanf 从缓冲区中读取格式化字符串函数

**函数原型:** int sscanf(char \*string, char \*format[, argument, ...]);

**头文件:** #include <stdio.h>

**是否是标准函数:** 是

**函数功能:** 将 string 指定的数据读到 argument 所指定的位置。其中, 参数 argument 是与 format 格式要求相符合的变量指针。也就是说, 如果 format 指定的格式为 "%d", 则 argument 就必须是整型变量的指针。这里应该注意, sscanf 函数与 scanf、fscanf 函数不同, 前者是从指定的缓冲区读格式化数据到新的缓冲区中, 而后者是从标准输入文件 (stdin) 和用户自定义文件中读取格式化数据到缓冲区中。

**返回值:** 成功返回已分配空间的数量, 返回 0 表示未使用分配空间, 返回 EOF 则表示出错。

#### 例程 10-37 应用 sscanf 函数读取格式化数据。

```
#include <stdio.h>
void main( void )
{
    char str[] = "1 2 3...";
    char s[81];
```



```

char c;
int i;
float fp;
/* 从缓冲区 str 中读取数据 */
sscanf( str, "%s", s );
sscanf( str, "%c", &c );
sscanf( str, "%d", &i );
sscanf( str, "%f", &fp );
/* 输出已读取的数据 */
printf( "String    = %s\n", s );
printf( "Character = %c\n", c );
printf( "Integer:  = %d\n", i );
printf( "Real:     = %f\n", fp );
}

```

#### 例程说明:

- (1) 首先开辟一个以 str 为首地址的缓冲区，并初始化其内容。
- (2) 应用 sscanf 函数从缓冲区 str 中读取数据。分别以格式要求"%s"、"%c"、"%d"、"%f"读取，并存入相应的变量中。
- (3) 输出已读取的数据。

## 10.35 tmpfile 创建临时文件函数

函数原型: FILE \*tmpfile(void);

头文件: #include<stdio.h>

是否是标准函数: 是

函数功能: 创建一个临时文件。该文件以 w+b（二进制读写）方式打开，当关闭时，该文件会被自动删除。

返回值: 返回指向临时文件的指针，如果文件打不开则返回 EOF。

#### 例程 10-38 创建一个临时文件。

```

#include <stdio.h>
#include <process.h>
int main(void)
{
    FILE *tempfp;
    tempfp = tmpfile();
    if (tempfp)
        printf("Temporary file be created!!\n");
    else
    {
        printf("Unable to create the temporary file!!\n");
        exit(1);
    }
    sleep(20);
    return 0;
}

```

#### 例程说明:

- (1) 首先应用 tmpfile 函数创建一个临时文件，并将文件指针赋值给 FILE 型变量 tempfp。



- (2) 如果创建临时文件成功, 则在终端显示: Temporary file be created!!
- (3) 如果创建不成功, 则显示: Unable to create the temporary file!!
- (4) 程序挂起 20 秒。

注意: 这里将程序挂起 20 秒的目的是为了让用户看到生成的临时文件。这是因为当程序执行完时, 系统会自动删除临时文件, 那样用户就感觉不到临时文件的创建了。当该程序运行时, 会在当前文件夹下生成一个临时文件。临时文件的作用是暂时存储程序运行过程中需要的数据, 当程序运行完毕时, 这些数据也就没有用了。

## 10.36 tmpnam 创建临时文件名函数

函数原型: `char *tmpnam(char *string);`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 创建一个临时文件名, 用以打开一个临时文件。

返回值: 创建成功返回指向该文件名的指针, 否则返回 NULL。

### 例程 10-39 创建一个临时文件名。

```
#include <string.h>
#include <stdio.h>
main()
{
    char tmp[10];
    tmpnam(tmp);
    printf("The temporary name is %s\n", tmp);
}
```

例程说明:

- (1) 首先定义一个字符型数组 `tmp`。
- (2) 调用 `tmpnam` 函数生成一个临时文件名。
- (3) 打印出该临时文件名。

注意: 应用函数 `tmpnam` 生成的临时文件名是不同于任何已存在文件名的有效文件名。本函数用于给临时文件创建文件名。

## 10.37 ungetc 把字符退回到输入流函数

函数原型: `int ungetc(char c, FILE *fp);`

头文件: `#include <stdio.h>`

是否是标准函数: 是

函数功能: 把字符 `c` 退回到 `fp` 所指向的文件流中, 并清除文件的结束标志。 `fp` 所指向

的文件既可以是用户自定义的文件，也可以是系统定义的标准文件。

返回值：成功返回字符 *c*，否则返回 EOF。

#### 例程 10-40 应用 ungetc 函数向标准输入文件退回字符。

```
#include <stdio.h>
#include <ctype.h>
int main( void )
{
    int i=0;
    char ch;
    puts("Input an integer followed by a char:");
    /*读取字符直到输入非数字或 EOF*/
    while((ch = getchar()) != EOF && isdigit(ch))
        i = 10 * i + ch - 48; /* 将 ASCII 码转换为整数值*/
    /*如果输入非数字，将其退回输入流*/
    if (ch != EOF)
        ungetc(ch, stdin);
    printf("i = %d, next char in buffer = %c\n", i, getchar());
    return 0;
}
```

#### 例程说明：

- (1) 首先从终端输入一个字符串，要求输入整型数字，并以非数字字符结尾。
- (2) 程序读取字符，直到输入非数字或 EOF 为止，并将数字字符串转换为整型数。

例如：将字符串"123"转换为整型数 123。

- (3) 然后将结尾的非数字字符退回到标准输入文件 (stdin)。
- (4) 显示退回到标准输入文件中的字符。

注意：所谓将结尾的非数字字符退回到标准输入文件，就是将数字字符串后面的那个非数字字符退回到标准输入文件中。例如输入的字符串为"123abc"，那么退回的字符就是 a。这样，程序将前面的字符串"123"转换为整型数 123，并存入变量 *i* 中。a 作为数字字符串输入的结束标志（因为 a 是数字字符之后的第一个非数字字符）被退回到标准输入文件 (stdin)。于是，再调用 `getchar` 函数读取的字符，就应该是刚刚退回到标准输入文件中的字符 a。因此，本段例程的执行结果为：

```
Input an integer followed by a char:
123abc
i = 123, next char in buffer = a
```

即输入的字符串中 123 为整型数，接下来的字符为 a。



# 字符处理函数

头文件 `ctype.h` 中定义了许多字符处理函数, 这些函数主要用来对 ASCII 字符进行判断和操作。本章将对这些函数进行介绍。

## 11.1 isalnum 检查字符是否是字母或数字

函数原型: `int isalnum( int c );`

头文件: `#include <ctype.h>`

是否是标准函数: 是

函数功能: 检查字符 `c` 是否是字母 (alpha) 或数字 (number)。

返回值: 是字母或数字返回 1, 否则返回 0。

### 例程 11-1 应用 isalnum 检查字符属性。

```
#include <stdio.h>
#include <ctype.h>
int main( void )
{
    char c, ch;
    scanf("%c", &c);
    ch = getchar();
    while(c != 'e') {
        if(isalnum(c))
            printf("This is a alpha or a number\n");
        else
            printf("This is a particulate character\n");
        scanf("%c", &c);
        ch = getchar();
    }
    return 1;
}
```

#### 例程说明:

- (1) 首先程序声明了两个字符型变量, 用以接收来自终端的字符。
- (2) 当用户输入的字符不是 'e' 而是字母或数字字符时, 就在屏幕上显示 "This is a alpha or a number" 提示信息。当用户输入的字符不是 'e' 也不是字母或数字字符时, 就在屏幕上显示 "This is a particulate character" 提示信息。
- (3) 当用户输入字符 'e' 时, 程序退出。

注意：本例程中，scanf 函数用以接收欲判断的字符，getchar 函数用以接收回车换行符。  
本例程的运行结果为：

```
a
This is a alpha or a number
2
This is a alpha or a number
#
This is a particulate character
e
```

## 11.2 isalpha 检查字符是否是字母

函数原型：int isalpha( int c );

头文件：#include <ctype.h>

是否是标准函数：是

函数功能：检查字符 c 是否是字母（alpha）。

返回值：是字母返回 1，否则返回 0。

### 例程 11-2 应用 isalpha 检查字符属性。

```
#include <stdio.h>
#include <ctype.h>
int main( void )
{
    char c, ch;
    scanf("%c",&c);
    ch=getchar();
    while(c!='e') {
        if(isalpha (c))
            printf("This is a alpha \n");
        else
            printf("This is not a alpha\n");
        scanf("%c",&c);
        ch=getchar();
    }
    return 1;
}
```

#### 例程说明：

本例程与例程 11-1 相似，但只判断输入的字符是否是字母，如果是字母，则在屏幕上显示"This is a alpha "提示信息，否则显示"This is not a alpha"提示信息。

本例程的运行结果为：

```
a
This is a alpha
3
This is not a alpha
$
This is not a alpha
e
```



## 11.3 isascii 检查字符是否是 ASCII 码

函数原型: `int isascii(int c);`

头文件: `#include <ctype.h>`

是否是标准函数: 是

函数功能: 检查字符 `c` 是否是 ASCII 码。

返回值: 是 ASCII 码返回 1, 否则返回 0。

### 例程 11-3 应用 isascii 检查字符属性。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    c='A';
    printf("%c:%s\n",c,isascii(c)?"yes":"no");
    c=0x7f;
    printf("%c:%s\n",c,isascii(c)?"yes":"no");
    c=0x80;
    printf("%c:%s\n",c,isascii(c)?"yes":"no");
    getchar();
    return 0;
}
```

例程说明:

本例程应用 `isascii` 函数判断字符'A'、0x7f、0x80 是否是 ASCII 码, 如果是, 显示"yes", 不是则显示"no"。本例程的运行结果是:

```
A:yes
Δ:yes
Ç:no
```

注意: ASCII 码是指 0x00~0x7F 之间的字符, 本例程中十六进制数 0x7f 的字符显示为 Δ, 属于 ASCII 码, 因此显示 yes; 0x80 的字符显示为 Ç, 不属于 ASCII 码, 因此显示 no。

## 11.4 iscntrl 检查字符是否是控制字符

函数原型: `int iscntrl(int c);`

头文件: `#include <ctype.h>`

是否是标准函数: 是

函数功能: 检查字符 `c` 是否是控制字符, 控制字符的 ASCII 码在 0~0x1F 之间。

返回值: 是控制字符返回 1, 否则返回 0。

### 例程 11-4 应用 iscntrl 检查字符属性。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
```

```
{
    char c,ch;
    printf("Input some character until contrl character\n");
    scanf("%c",&c);
    ch=getchar();
    while(!iscntrl(c)){
        scanf("%c",&c);
        ch=getchar();
    };
    return 0;
}
```

#### 例程说明:

本例程的功能为, 当输入的字符不是控制字符时, 可以一直输入下去, 一旦输入了控制字符, 程序结束。

本例程的运行结果为:

```
Input some character until contrl character
a
b
```

注意: 每输入一个字符时, 要以回车结束。

## 11.5 isdigit 检查字符是否是数字字符

函数原型: `int isdigit( int c );`

头文件: `#include <ctype.h>`

是否是标准函数: 是

函数功能: 检查字符 `c` 是否是数字字符 (0~9)。

返回值: 是数字字符返回 1, 否则返回 0。

#### 例程 11-5 应用 isdigit 函数统计字符串中数字的个数。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char ch;
    int i=0;
    ch=getchar();
    while(ch!=EOF){
        if(isdigit(ch))i++;
        ch=getchar();
    }
    printf("%d",i);
}
```

#### 例程说明:

(1) 首先程序中设置字符型变量 `ch`, 用以接收输入的字符; 设置整型变量 `i`, 用以统计输入的字符串中数字的个数, 并初始化 `i=0`。

(2) 当输入的字符不是 `EOF` 时, 程序循环执行, 并应用 `isdigit` 函数判断用户输入的字符是否是数字字符, 如果是则在变量 `i` 上加 1。



(3) 最后显示输入的字符串中数字的个数。

本例程的运行结果为：

```
abc123def567ghi^Z
6
```

注意：利用 Ctrl+Z 组合键输入的字符就是 EOF。

## 11.6 isxdigit 检查字符是否是十六进制数字字符

函数原型：int isxdigit(int c);

头文件：#include <ctype.h>

是否是标准函数：是

函数功能：检查字符 c 是否为十六进制数字。

返回值：当 c 为 A~F, a~f 或 0~9 之间的十六进制数字时，返回非零值，否则返回 0。

**例程 11-6** 应用 isxdigit 函数检查字符属性。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char c;
    c='f';
    printf("%c:%s\n",c,isxdigit(c)?"yes":"no");
    c='1';
    printf("%c:%s\n",c,isxdigit(c)?"yes":"no");
    c='$';
    printf("%c:%s\n",c,isxdigit(c)?"yes":"no");
    getchar();
    return 0;
}
```

**例程说明：**

本例程应用 isxdigit 函数判断字符 'f'、'1'、'\$' 是否是十六进制数字，如果是，显示 "yes"，是则显示 "no"。本例程的运行结果是：

```
f:yes
1:yes
$:no
```

## 11.7 isgraph 检查字符是否是可打印字符（不含空格）

函数原型：int isgraph(int c);

头文件：#include <ctype.h>

是否是标准函数：是

**函数功能：**检查字符 *c* 是否是除了空格符外的可打印字符，其 ASCII 码在 0x21~0x7e 之间。

**返回值：**是除了空格符外的可打印字符则返回 1，否则返回 0。

#### 例程 11-7 应用 isgraph 函数判断可打印字符。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    c='A';
    printf("%c:%s\n",c,isgraph(c)?"yes":"no");
    c=' ';
    printf("%c:%s\n",c,isgraph(c)?"yes":"no");
    c=0x7f;
    printf("%c:%s\n",c,isgraph(c)?"yes":"no");
    getchar();
    return 0;
}
```

#### 例程说明：

本例程与例程 11-6 相似，应用 isgraph 函数判断字符'A'、' '、0x7f 是否是除了空格符外的可打印字符。如果是，显示"yes"，不是则显示"no"。本例程的运行结果是：

```
A:yes
:no
Δ:no
```

## 11.8 isprint 检查字符是否是可打印字符（含空格）

**函数原型：**int isprint(int c);

**头文件：**#include <ctype.h>

**是否是标准函数：**是

**函数功能：**检查字符 *c* 是否为可打印字符（含空格），其 ASCII 码在 0x20~0x7e 之间。

**返回值：**是可打印字符返回 1，否则返回 0。

#### 例程 11-8 应用 isprint 函数判断可打印字符。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    c='A';
    printf("%c:%s\n",c,isprint(c)?"yes":"no");
    c=' ';
    printf("%c:%s\n",c,isprint(c)?"yes":"no");
    c=0x7f;
    printf("%c:%s\n",c,isprint(c)?"yes":"no");
    getchar();
    return 0;
}
```



**例程说明:**

本例程与例程 11-6 相似,应用 isprint 函数判断字符'A'、' '、0x7f 是否是可打印字符(包括空格)。如果是,显示"yes",不是则显示"no"。本例程的运行结果是:

```
A:yes
:yes
Δ:no
```

## 11.9 ispunct 检查字符是否是标点字符

函数原型: int ispunct(int c);

头文件: #include <ctype.h>

是否是标准函数: 是

函数功能: 检查字符 c 是否是除字母、数字、空格之外的可打印字符,也就是检查字符 c 是否是标点字符。

返回值: 当 c 为标点字符时返回 1, 否则返回 0。

**例程 11-9 应用 ispunct 函数判断标点字符。**

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
int main(void)
{
    char s[]="He said:Oh!Very well!";
    int i;
    printf("%s\n",s);
    for(i=0;i<strlen(s);i++)
    {
        if(ispunct(s[i])) printf("^");
        else printf(".");
    }
    return 0;
}
```

**例程说明:**

(1) 首先将字符串"He said:Oh!Very well!"存入以 s 为首地址的缓冲区中,并在屏幕上显示该字符串。

(2) 循环检查该字符串中的每个字符,并在屏幕上显示的字符串下方做出标记,如果不是标点字符,打印 "."; 如果是标点字符,打印 "^"。本例程的运行结果是:

```
He said:Oh!Very well!
. . . . . ^ . . . . . ^ . . . . . ^
```

## 11.10 islower 检查字符是否是小写字母

函数原型: int islower(int c);

头文件: #include <ctype.h>

是否是标准函数：是

函数功能：检查字符 *c* 是否是小写字母 (a~z)。

返回值：当 *c* 为小写字母时返回 1，否则返回 0。

#### 例程 11-10 应用 islower 函数统计字符串中的小写字母个数。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char ch;
    int i=0;
    ch=getchar();
    while(ch!=EOF){
        if(islower(ch))i++;
        ch=getchar();
    }
    printf("%d",i);
    getchar();
    return 0;
}
```

例程说明：

本例程与例程 11-5 相似，先输入一串任意的字符，然后应用 islower 函数统计字符串中的小写字母个数。最后在屏幕上显示出小写字母的个数。本例程的运行结果是：

```
djcvGGJH4623^Z
4
```

注意：^Z 是 Ctrl+Z 组合键的屏幕显示，即结束标志 EOF。

## 11.11 isupper 检查字符是否是大写字母

函数原型：int isupper(int c);

头文件：#include<ctype.h>

是否是标准函数：是

函数功能：检查字符 *c* 是否是大写字母 (A~Z)。

返回值：当 *c* 为大写字母时返回 1，否则返回 0。

#### 例程 11-11 应用 isupper 函数统计字符串中的大写字母个数。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char ch;
    int i=0;
    ch=getchar();
    while(ch!=EOF){
        if(isupper(ch))i++;
        ch=getchar();
    }
    printf("%d",i);
}
```



```
    getchar();  
    return 0;  
}
```

例程说明:

本例程与例程 11-5 相似，利用函数 `isupper` 统计输入的字符串中大写字母的个数。最后在屏幕上显示出大写字母的个数。本例程的运行结果是：

ABCDEabcFG123^Z  
7

### 11.12 isspace 检查字符是否是空格符

函数原型: `int isspace(int c);`

头文件: `#include<ctype.h>`

是否是标准函数：是

**函数功能：**检查字符 c 是否为空格符 space、制表符 tab 或是换行符。空格符 space 的 ASCII 码为 32，制表符 tab 的 ASCII 码为 9，换行符的 ASCII 码则为 10。

**返回值：**当 c 为空格符或制表符时，返回 1，否则返回 0。

### 例程 11-12 应用 isspace 函数转换空格符、制表符和换行符。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char s[]="space |NewLine\n|table\t|";
    int i;
    printf("%s",s);
    printf("\n");
    for(i=0;i<strlen(s);i++)
    {
        if(isspace(s[i])) putchar('.');
        else putchar(s[i]);
    }
    getchar();
    return 0;
}
```

例程说明:

(1) 首先将字符串"space\nNewLine\nltable\t"存入以 s 为首地址的缓冲区中,并在屏幕上显示该字符串。其中' '、'\n'、'\t'分别为空格符、换行符、制表符,在屏幕上显示其字符原样。

(2) 然后通过 isspace 函数检测出该字符串中的这些空格符、换行符、制表符, 将其转换为' '字符, 并输出到终端屏幕。

注意: 本例程并没有改变原字符串数组中的存储内容, 只是在输出时将字符串中的空格符、换行符、制表符转换为' '字符并输出到终端屏幕。

本例程的运行结果是:

```
space |NewLine
|table |
space.|NewLine.|table.|
```

## 11.13 toascii 将字符转换为 ASCII 码

函数原型: `int toascii(int c);`

头文件: `#include <ctype.h>`

是否是标准函数: 是

函数功能: 将 `c` 转化为相应的 ASCII 码。

返回值: 返回转换后的数值, 也就是转换后的 ASCII 码。

**例程 11-13** 应用 `toascii` 函数将整型数字转换为相应的 ASCII 码。

```
#include <stdio.h>
#include <ctype.h>
main()
{
    int s[]={1,2,3,4,5,6};
    int i;
    for(i=0;i<6;i++)
    {
        printf("%d",s[i]);
    }
    printf("\n");
    for(i=0;i<6;i++)
    {
        putchar(toascii(s[i]));
    }
    getchar();
    return 0;
}
```

例程说明:

(1) 首先在整型数组中存入 1~6 6 个整型数字, 并将其显示在终端屏幕上。

(2) 循环地将数组中的每个数字转换为其对应的 ASCII 码, 并将其以字符的形式显示在终端屏幕上。本例程的运行结果为:

```
123456
000000
```

## 11.14 tolower 将大写字母转换为小写字母

函数原型: `int tolower(int c);`

头文件: `#include <ctype.h>`

是否是标准函数: 是

函数功能: 将 `c` 转化为相应的小写字母。

返回值: 如果 `c` 为大写英文字母, 则返回对应的小写字母; 否则返回原来的值。



**例程 11-14** 应用 tolower 函数将大写字母转换为小写字母。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char str[]="This Is A Test!";
    int i;
    printf("%s\n",str);
    for(i=0;i<strlen(str);i++)
    {
        putchar(tolower(str[i]));
    }
    getchar();
    return 0;
}
```

**例程说明:**

(1) 首先将字符串"This Is A Test!"存入以 str 为首地址的缓冲区中,并将该字符串显示在终端屏幕上。

(2) 应用 tolower 函数将该字符串中大写字母转换为小写字母,并输出在终端屏幕上。

注意: 本例程将字符串中大写字母转换为小写字母并输出,但并不改变原数组中的内容,只是在输出时将大写字母转换为小写字母,而本身是小写字母的字符或非字母字符,则返回原值。

本例程的运行结果是:

```
This Is A Test!
this is a test!
```

## 11.15 toupper 将小写字母转换为大写字母

函数原型: int toupper(int c);

头文件: #include<ctype.h>

是否是标准函数: 是

函数功能: 将 c 转化为相应的大写字母。

返回值: 如果 c 为小写英文字母,则返回对应的大写字母;否则返回原来的值。

**例程 11-15** 应用 toupper 函数将小写字母转换为大写字母。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char str[]="This Is A Test!";
    int i;
    printf("%s\n",str);
    for(i=0;i<strlen(str);i++)
    {
        putchar(toupper(str[i]));
    }
    getchar();
}
```

```
    return 0;  
}
```

### 例程说明:

本例程与例程 11-14 相似, 利用 `toupper` 函数将字符串中的小写字母转换为大写字母, 并输出到终端。本例程的运行结果为:

```
This Is A Test!  
THIS IS A TEST!
```



# 字符串处理函数

在 C 语言中字符串是一个重要的概念，字符串处理函数是针对字符串进行操作（字符串比较、字符串拷贝等）的一系列库函数，主要定义在 `string.h` 头文件中。本章将介绍一些常用的字符串处理函数，其中包括一些非标准函数。

## 12.1 strcat 字符串连接函数

函数原型：char \*strcat (char \*dest,char \*src);

头文件：#include<string.h>

是否是标准函数：是

函数功能：将两个字符串连接合并成一个字符串，也就是把字符串 src 连接到字符串 dest 后面，连接后的结果放在字符串 dest 中。

返回值：指向字符串 dest 的指针。

### 例程 12-1 应用 strcat 连接字符串。

```
#include <string.h>
#include <stdio.h>
int main( )
{
    char dest[20]={" "};
    char *hello = "hello ", *space = " ", *world = "world";
    strcat(dest, hello);
    strcat(dest, space);
    strcat(dest, world);
    printf("%s\n", dest);
    getch();
    return 0;
}
```

#### 例程说明：

(1) 首先程序声明了一个字符数组和三个字符串变量，将字符数组 dest 初始化为空串，其余三个字符串变量分别赋予初值。

(2) 程序通过调用 strcat 函数实现字符串的连接，首先将字符串"hello"添加到字符数组 dest 的末端，此时字符数组 dest 的值由空串变为"hello"，然后继续调用两次 strcat 函数，依次将字符串 space 和字符串 world 连接到字符数组 dest 的末端，从而完成整个字符串的连接操作。

(3) 最后将最终的结果输出。



本例程的运行结果是：

```
hello world
```

注意：本例程中，开始时把字符数组 `dest` 初始化为空是必要的，对声明的变量进行初始化是一个很好的习惯。如果不对字符数组 `dest` 进行初始化，程序运行时会产生错误，有兴趣的读者可以试试未初始化时程序的输出结果。

## 12.2 strncat 字符串连接函数

函数原型：char \*strncat (char \*dest, char \*src, int n);

头文件：#include <string.h>

是否是标准函数：是

函数功能：将一个字符串的子串连接到另一个字符串末端，也就是把字符串 `src` 的前 `n` 个字符连接到字符串 `dest` 后面，连接后的结果放在字符串 `dest` 中。

返回值：指向字符串 `dest` 的指针。

### 例程 12-2 应用 strncat 连接字符串子串。

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char dest[30]={""};
    char *favorite = "I love", *tabs = "\t\n", *language = "C++";
    strncat(dest, favorite,6);
    strncat(dest, tabs,1);
    strncat(dest, language,1);
    printf("%s\n", dest);
    getch();
    return 0;
}
```

例程说明：

(1) 首先程序声明了一个字符数组和三个字符串变量，将字符数组 `dest` 初始化为空串，其余三个字符串变量分别赋予初值，其中字符串 `"tans"` 由制表符和换行符两个字符组成。

(2) 程序通过调用 `strncat` 函数实现字符串子串的连接，首先将字符串 `"favorite"` 的前 6 个字符添加到字符数组 `dest` 的末端，其效果与直接调用 `strcat` 函数相同，然后继续调用两次 `strncat` 函数，依次将字符串 `"tabs"` 和字符串 `"language"` 的首字符连接到字符数组 `dest` 的末端，从而完成整个字符串子串的连接操作。

(3) 最后将最终的结果输出，由于未将字符串 `"tabs"` 中的换行符添加到字符数组 `dest` 中，因此所有输出结果应在同一行。

本例程的运行结果是：

```
I love C
```



注意：本例程中，字符串"tabs"中的内容并不是一般的字符，而是两个转义说明符构成的特殊字符，C 语言内部在处理过程中遇到转义说明符时会做特殊处理，本例中会将'\t'看作制表符，将'\n'看作换行符。

## 12.3 strcmp 字符串比较函数

函数原型：int strcmp(char \*str1, char \*str2);

头文件：#include <string.h>

是否是标准函数：是

函数功能：比较两个字符串的大小，也就是把字符串 str1 和字符串 str2 从首字符开始逐字符地进行比较，直到某个字符不相同或比较到最后一个字符为止，字符的比较为 ASCII 码的比较。

返回值：若字符串 str1 大于字符串 str2，返回结果大于零；若字符串 str1 小于字符串 str2，返回结果小于零；若字符串 str1 等于字符串 str2，返回结果等于零。

### 例程 12-3 应用 strcmp 比较字符串大小。

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char *str1 = "Canada", *str2 = "China", *str3 = "china";
    int result;
    result = strcmp(str1, str2);
    if (result < 0)
        printf("%s is less than %s", str1, str2);
    else
        printf("%s is not less than %s", str1, str2);
    printf("\n");
    result = strcmp(str2, str3);
    if (result < 0)
        printf("%s is less than %s", str2, str3);
    else
        printf("%s is not less than %s", str2, str3);
    getch();
    return 0;
}
```

#### 例程说明：

(1) 首先程序声明了三个字符串变量并分别赋予了初值，注意字符串 str2 和字符串 str3 的区别在于首字母是否大写，整型变量 result 用于记录字符串的比较结果。

(2) 程序通过调用 strcmp 函数比较字符串 str1 和字符串 str2，在首字符相同的情况下第二个字符'a'的 ASCII 码小于'h'的 ASCII 码，因此比较结果为字符串 str1 小于字符串 str2，返回结果小于零。第二次调用 strcmp 函数比较字符串 str2 和字符串 str3，由于在 ASCII 码表中小写字母在后，小写字母的 ASCII 码大于大写字母，即'C'小于'c'，因此比较结果为字符串 str2 小于字符串 str3，返回结果小于零。

(3) 最后将最终的结果输出。为了使输出结果一目了然，在两次比较的中间使用 printf



函数输出了一个换行。

本例程的运行结果是：

```
Canada is less than China
China is less than china
```

注意：本例程中，字符串的比较结果为首个两个不等字符之间 ASCII 码的差值，如果我们将第一次比较的结果 result 输出，应该是'a'的 ASCII 码与'h'的 ASCII 码的差值，有兴趣的读者可以试试输出结果。

## 12.4 strncmp 字符串子串比较函数

函数原型：int strncmp (char \*str1, char \* str2, int n);

头文件：#include <string.h>

是否是标准函数：是

函数功能：比较两个字符串子串的大小，也就是把字符串 str1 的前 n 个字符组成的子串和字符串 str2 的前 n 个字符组成的子串进行比较。

返回值：若字符串 str1 前 n 个字符组成的子串大于字符串 str2 前 n 个字符组成的子串，返回结果大于零；若字符串 str1 前 n 个字符组成的子串小于字符串 str2 前 n 个字符组成的子串，返回结果小于零；若字符串 str1 前 n 个字符组成的子串等于字符串 str2 前 n 个字符组成的子串，返回结果等于零。

### 例程 12-4 应用 strncmp 比较字符串子串大小。

```
#include <string.h>
int main(void)
{
    char *str1="Hello World";
    char *str2="Hello C Programme";
    int result;
    result=strncmp(str1,str2,5);
    if(!result)
        printf("%s is identical to %s in the first 5 words",str1,str2);
    else if(result<0)
        printf("%s is less than %s in the first 5 words",str1,str2);
    else
        printf("%s is great than %s in the first 5 words",str1,str2);
    printf("\n");
    result=strncmp(str1,str2,10);
    if(!result)
        printf("%s is identical to %s in the first 10 words",str1,str2);
    else if(result<0)
        printf("%s is less than %s in the first 10 words",str1,str2);
    else
        printf("%s is great than %s in the first 10 words",str1,str2);
    getch();
    return 0;
}
```



**例程说明:**

(1) 首先程序声明了两个字符串变量并分别赋予了初值, 整型变量 `result` 用于记录字符串子串的比较结果。

(2) 程序通过调用 `strncmp` 函数比较字符串 `str1` 和字符串 `str2` 的前 5 个字符组成的子串, 由于两个字符串的前 5 个字符相同, 因此两个子串的比较结果应为相等, 返回结果为零。然后将比较结果输出。

(3) 程序第二次调用 `strncmp` 函数比较字符串 `str2` 和字符串 `str3` 的前 10 个字符组成的子串, 由于从第 7 个字符开始出现了不等的情况, 分别为 'w' 和 'C', 根据 ASCII 码表中小写字母在后, 即 'w' 的 ASCII 码大, 返回结果大于零。最后输出比较结果。

(4) 输出时显示比较结果并指明比较范围。

本例程的运行结果是:

```
Hello World is identical to Hello C Programme in the first 5 words
Hello World is great than Hello C Programme in the first 10 words
```

注意: 本例程中, 要注意子串比较的过程中, 子串的大小应不小于零且不超过字符串的长度。虽然子串的长短参数不会产生编译时的错误和影响最终结果的输出, 但在比较前检查比较范围是一个很好的习惯。

## 12.5 strcpy 字符串拷贝函数

**函数原型:** `char * strcpy (char *dest, char * src);`

**头文件:** `#include <string.h>`

**是否是标准函数:** 是

**函数功能:** 对字符串的拷贝, 也就是把字符串 `src` 中的内容复制到字符串 `dest` 中, 使两个字符串的内容相同。

**返回值:** 指向字符串 `dest` 的指针。

### 例程 12-5 应用 strcpy 实现字符串拷贝。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char dest[20] = {" "};
    char *src = "Hello World";
    int result;
    strcpy(dest, src);
    printf("%s\n", dest);
    result = strcmp(dest, src);
    if (!result)
        printf("dest is equal to src");
    else
        printf("dest is not equal to src");
    getch();
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了一个字符串和一个字符数组并分别赋予了初值，整型变量 `result` 用于记录字符串子串的比较结果。

(2) 程序通过调用 `strcpy` 函数将字符串 `src` 中的内容复制到字符数组 `dest` 中，使得两者具有相同的内容。为了验证两个变量中的内容是否真的一样，通过调用 `strcmp` 对两个字符串中的内容进行比较。

(3) 最后将复制结果和比较结果输出。

本例程的运行结果是：

```
Hello World
dest is equal to src
```

注意：本例程中，向字符数组中赋值时要保证字符数组中有足够的空间，虽然有时候即便空间不够也会打印出正确的结果，但随着程序的运行，不能保证超出下标范围的部分还能以正确的形式存在。

## 12.6 strncpy 字符串子串拷贝函数

函数原型：char \* strncpy (char \*dest,char \* src, int n);

头文件：#include<string.h>

是否是标准函数：是

函数功能：实现字符串子串的拷贝，也就是把字符串 `src` 中的前 `n` 个字符复制到字符串 `dest` 中。

返回值：指向字符串 `dest` 的指针。

### 例程 12-6 应用 strncpy 实现字符串子串的拷贝。

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char dest[20]={""};
    char *src1="Hello World",*src2="Aloha";
    strncpy(dest,src1,5);
    strncpy(dest,src2,5);
    if(!strcmp(dest,src1))
        printf("dest is equal to src1");
    else if(!strcmp(dest,src2))
        printf("dest is equal to src2");
    else
        printf("dest is %s",dest);
    printf("%s\n", dest);
    getch();
    return 0;
}
```



**例程说明:**

(1) 首先程序声明了两个字符串和一个字符数组并分别赋予了初值, 本例中省去了用于记录比较结果的 `result` 变量。

(2) 程序通过调用 `strncpy` 函数将字符串 `src1` 中的前 5 个字符组成的子串复制到字符数组 `dest` 中, 然后再次调用 `strncpy` 函数将字符串 `src2` 中的前 5 个字符组成的子串复制到字符数组 `dest` 中。通过调用一系列的 `strcmp` 函数, 从而达到验证 `dest` 变量中的最终内容的目的。

(3) 最终的字符串 `dest` 中的内容到底是什么呢, 是 "Hello"、"Aloha" 还是 "HelloAloha"? 通过第一次调用 `strncpy` 函数, 字符串 `dest` 中的内容由空串变成 "Hello", 再次调用 `strncpy` 函数则会从字符串 `dest` 的下标为零处逐一覆盖, 也就是 "Aloha" 覆盖了原来的 "Hello", 并不是将 "Aloha" 添加到末端。

(4) 最后将复制结果和验证结果输出。

本例程的运行结果是:

```
dest is equal to src2 Aloha
```

注意: 本例程中, 在检验字符串 `dest` 的内容时, `if` 判断中并没有使用像例程 12-5 中的 `result` 变量, 我们只关心比较的结果是否为零, 所以直接通过将函数作为判断条件, 从而利用函数的返回值进行判断, 这种判断方法更为简单、有效。

## 12.7 strlen 计算字符串长度函数

函数原型: `int strlen (char *str);`

头文件: `#include <string.h>`

是否是标准函数: 是

函数功能: 求字符串的长度, 也就是求字符串 `str` 中有多少个字符。

返回值: 字符串 `str` 字符的个数。

### 例程 12-7 应用 strlen 求字符串长度。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char src1[3]={" "},src2[10]="Hello";
    char *src3="Hello";
    printf("%d\n",strlen(src1));
    printf("%d\n",strlen(src2));
    printf("%d\n",strlen(src3));
    getch();
    return 0;
}
```



**例程说明:**

(1) 首先程序声明了一个字符串和两个字符数组并分别赋予了初值, 将字符串 `src3` 与字符数组 `src2` 赋予相同的初值。

(2) 程序通过调用 `strlen` 函数分别求出三个变量字符的长度。在求字符长度时, 返回的结果是有效字符的个数, 因此虽然字符数组 `src1` 由十个字符型变量组成, 但初值为空串, 因此长度为零, 并不等于数组长度。由于字符串 `src3` 与字符数组 `src2` 赋予相同的初值, 因此两者长度相同。

(3) 最后将字符串或字符数组的长度值输出。

本例程的运行结果是:

```
0
5
5
```

注意: 本例程中, 如果将字符数组 `src2` 复制到字符数组 `src1` 中, 并不会产生任何编译错误, 但是程序运行会产生不可预知的结果, 有兴趣的读者可以在完成拷贝后将三个变量的长度输出。

## 12.8 strchr 字符串中字符首次匹配函数

函数原型: `char *strchr(char *str, char c);`

头文件: `#include <string.h>`

是否是标准函数: 是

函数功能: 在字符串中查找给定字符的第一次匹配, 也就是在字符串 `str` 中查找字符 `c` 第一次出现的位置。

返回值: 第一次匹配位置的指针。

### 例程 12-8 应用 `strchr` 匹配字符串中字符。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[15] = {" "};
    char *ptr, c = 'r';
    strcpy(str, "Hello World");
    ptr = strchr(str, c);
    if (ptr)
        printf("The character %c is at position: %d\n", c, ptr-str);
    else
        printf("The character was not found\n");
    strcpy(str, "Aloha");
    if (ptr)
        printf("The character %c is at position: %d\n", c, ptr-str);
    else
        printf("The character was not found\n");
    getch();
    return 0;
}
```



**例程说明：**

(1) 首先程序声明了一个字符串、一个字符数组以及一个字符，并对字符型变量赋予了要查找的值。

(2) 程序通过调用 `strcpy` 赋予字符数组一个值，然后调用 `strchr` 函数在字符数组中查找第一次与字符型变量 `c` 匹配的字符，也就是查找第一个 'r' 字符，返回的结果为指向第一个匹配字符的指针。根据返回值输出匹配结果。

(3) 程序第二次通过调用 `strcpy` 赋予字符数组一个值，然后调用 `strchr` 函数在字符数组中查找第一次与字符型变量 `c` 匹配的字符，也就是查找第一个 'r' 字符，最后根据返回值输出匹配结果。

(4) 第二次匹配已经对字符串重新赋值，新的字符串似乎应该是 "Aloha"，从而没有与 'r' 匹配的字符，但实际的运行结果却不是。这是因为在重新赋值时，虽然 "Aloha" 将 "Hello" 覆盖掉，但是后面的字符仍然保留在数组中，因此匹配时得到的结果仍与第一次匹配结果相同。

(5) 对结果进行输出时，如果匹配成功，输出匹配的字符在数组中的位置；如果匹配不成功，则显示没有找到。

本例程的运行结果是：

```
The character r is at position: 8
The character r is at position: 8
```

注意：本例程中，字符串中字符匹配的返回值是指向匹配位置的指针，获取到该指针后，与数组的头指针做减法，也就是与数组变量名做减法，就可以获得指针在数组中对应的下标。

## 12.9 strchr 字符串中字符末次匹配函数

函数原型：char \*strchr(char \*str, char c);

头文件：#include <string.h>

是否是标准函数：是

函数功能：在字符串中查找给定字符的最后一次匹配，也就是在字符串 `str` 中查找字符 `c` 最后一次出现的位置。

返回值：最后一次匹配位置的指针。

### 例程 12-9 应用 strchr 匹配字符串中字符。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[15]={" "};
    char *ptr, c = 'o';
    strcpy(str, "Hello World");
    ptr = strchr(str, c);
    if (ptr)
        printf("The first character %c is at position: %d\n", c, ptr-str);
}
```



```
else
    printf("The character was not found\n");
ptr = strchr(str, c);
if (ptr)
    printf("The last character %c is at position: %d\n", c, ptr-str);
else
    printf("The character was not found\n");
getch();
return 0;
}
```

#### 例程说明:

(1) 首先程序声明了一个字符串、一个字符数组以及一个字符, 并对字符型变量赋予了要查找的值。

(2) 程序通过调用 `strcpy` 赋予字符数组一个值, 然后调用 `strchr` 函数在字符数组中查找第一次与字符型变量 `c` 匹配的字符, 也就是查找最后一个 'o' 字符。返回的结果为指向第一个匹配字符的指针, 根据返回值输出匹配结果。

(3) 然后程序调用 `strrchr` 函数在字符数组中查找最后一次与字符型变量 `c` 匹配的字符, 也就是查找最后一个 'o' 字符, 最后根据返回值输出匹配结果。

(4) 字符数组中有两个 'o' 字符, 因此调用 `strchr` 函数应该返回第一个 'o' 字符的指针, 调用 `strrchr` 函数应该返回最后一个 'o' 字符的指针。

(5) 对结果进行输出时, 如果匹配成功, 输出匹配的字符在数组中的位置; 如果匹配不成功, 则显示没有找到。

本例程的运行结果是:

```
The first character o is at position: 4
The last character o is at position: 7
```

注意: 本例程中, 如果字符串中只有一个 'o' 字符, 那么无论调用哪种字符串中的字符匹配函数都会返回相同的结果。

## 12.10 strspn 字符集匹配函数

函数原型: `int strspn(char *str1, char *str2);`

头文件: `#include <string.h>`

是否是标准函数: 是

函数功能: 在字符串中查找第一个不属于字符集的下标, 即从开始有多少个字符属于字符集。具体说是在字符串 `str1` 中查找第一个不属于字符集 `str2` 中任何一个字符的下标, 也就是字符串 `str1` 中属于字符集 `str2` 中的字符个数。

返回值: 所找到的字符串中段的长度。

#### 例程 12-10 应用 `strspn` 匹配字符串中字符集。

```
#include <string.h>
#include <stdio.h>
int main(void)
```



```
{
    char *str1="cabbage",*str2="potato";
    char *str= "abc";
    int result;
    result = strspn(str1,str);
    if(result)
        printf("The first %d is congruent\n",result);
    else
        printf("No character is congruent");
    result = strspn(str2,str);
    if(result)
        printf("The first %d is congruent\n",result);
    else
        printf("No character is congruent");
    getch();
    return 0;
}
```

#### 例程说明:

(1) 首先程序声明了三个字符串并分别赋予初值,其中最后一个变量是用于匹配的字符集。

(2) 程序通过调用 `strspn` 进行字符集的匹配,它从字符串 `str1` 中的第一个字符开始检查是不是属于字符串 `str` 中的任意字符,如果属于就继续匹配,直到匹配不成功。本例中会发现直到遇到字符'g'才匹配失败,因为字符'g'不属于字符串 `str` 中的任何一个字符。然后输出匹配结果。

(3) 程序第二次通过调用 `strspn` 对字符串 `str2` 进行字符集匹配,发现第一个字符就不属于字符集 `str` 中的任意字符。

(4) 输出匹配结果是显示前多少个字符匹配成功。

本例程的运行结果是:

```
The first 5 is congruent
No character is congruent
```

注意: 本例程中,进行字符集匹配时,待匹配的字符只要是字符集中的任意字符就匹配成功,要明确区分其余字符串匹配的不同。

## 12.11 strcspn 字符集逆匹配函数

函数原型: `int strcspn(char *str1, char *str2);`

头文件: `#include <string.h>`

是否是标准函数: 是

函数功能: 在字符串中查找第一个属于字符集的下标,即从开始有多少个字符不属于字符集,具体说是在字符串 `str1` 中查找第一个属于字符集 `str2` 中任何一个字符的下标,也就是字符串 `str1` 中不属于字符集 `str2` 中的字符个数。

返回值: 所找到的字符串中段的长度。



**例程 12-11 应用 strcspn 逆匹配字符串中字符集。**

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char *str1="tomato",*str2="carrot";
    char *str= "abc";
    int result;
    result = strcspn(str1,str);
    if(result)
        printf("The first %d is congruent\n",result);
    else
        printf("No character is congruent\n");
    result = strcspn(str2,str);
    if(result)
        printf("The first %d is congruent\n",result);
    else
        printf("No character is congruent\n");
    getch();
    return 0;
}
```

**例程说明:**

(1) 首先程序声明了三个字符串并分别赋予初值,其中最后一个变量是用于逆匹配的字符集。

(2) 程序通过调用 strcspn 进行字符集的逆匹配,它从字符串 str1 中的第一个字符开始检查是不是属于字符串 str 中的任意字符,如果不属于就继续逆匹配,直到逆匹配不成功。本例中会发现直到遇到字符'a'才逆匹配失败,因为字符'a'属于字符串 str 中的某个字符。然后输出匹配结果。

(3) 程序第二次通过调用 strcspn 对字符串 str2 进行字符集逆匹配,发现第一个字符即属于字符集 str 中的某字符。

(4) 输出匹配结果是显示前多少个字符逆匹配成功。

本例程的运行结果是:

```
The first 3 is congruent
No character is congruent
```

注意: 本例程中,字符集逆匹配与字符集匹配两者的匹配方式截然相反。字符串匹配是当字符串中字符等于字符集中任意字符时匹配成功,字符串逆匹配是当字符串中字符不等于字符集中任意字符时匹配成功。

## 12.12 strpbrk 字符集字符匹配函数

函数原型: char \*strpbrk(char \*str1, char \*str2);

头文件: #include<string.h>

是否是标准函数: 是

函数功能: 在字符串中查找第一个属于字符集的字符位置,也就是在字符串 str1 中查找



第一个属于字符集 str2 中任意字符的位置。

返回值：返回第一个匹配字符的指针。

### 例程 12-12 应用 strpbrk 匹配字符集字符。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str1="There are 5 pigs in the hogpen";
    char *str2="0123456789";
    char *result;
    result = strpbrk(str1,str2);
    if(result)
        printf("%s\n",result++);
    else
        printf("There are no numbers any more");
    result = strpbrk(result,str2);
    if(result)
        printf("%s\n",result++);
    else
        printf("There are no numbers any more");
    getch();
    return 0;
}
```

#### 例程说明：

(1) 首先程序声明了三个字符串变量并给前两个变量赋予初值，其中字符指针 result 用于记录匹配字符的位置。

(2) 程序通过调用 strcspn 进行字符集字符的匹配，它从字符串 str1 中查找第一个属于字符集 str2 中任意字符的字符。本例中是在字符串 str1 中查找第一个数字字符，如果匹配成功，则获得指向第一个数字字符的指针，利用该返回值输出匹配结果。

(3) 然后在上一次匹配字符的下一个字符作为首字符的子串中继续匹配，程序第二次通过调用 strspn，并用同样的输出方式输出匹配结果，如没有数字字符可以获得，则匹配失败。

(4) 输出匹配结果时并没有将匹配的字符输出，而是以匹配字符作为第一个字符的字符串子串输出。

本例程的运行结果是：

```
5 pigs in the hogpen
There are no numbers any more
```

注意：本例程中，匹配成功时结果的输出由于获得了匹配成功的字符的指针，因此可以利用该指针输出字符串的子串，利用自增操作符移动一个位置后又可以对尚未匹配的子串继续进行下一次匹配。

## 12.13 strstr 字符串匹配函数

函数原型：char \*strstr(char \*str1, char \*str2);

头文件：#include<string.h>



是否是标准函数：是

函数功能：在字符串中查找另一个字符串首次出现的位置，也就是在字符串 `str1` 中查找第一次出现字符串 `str2` 的位置。

返回值：返回第一次匹配字符串的指针。

### 例程 12-13 应用 `strstr` 匹配字符串。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str1 = "Borland International", *str2 = "nation";
    char *result;
    result = strstr(str1, str2);
    if (result)
        printf("The substring is: %s\n", result);
    else
        printf("Not found the substring");
    getch();
    return 0;
}
```

例程说明：

(1) 首先程序声明了三个字符串变量并对前两个变量赋予初值，其中字符指针 `result` 用于记录匹配字符串的位置。

(2) 程序通过调用 `strstr` 进行字符串匹配，查找字符串 `str1` 中首次出现字符串 `str2` 的位置，返回匹配结果。

(3) 输出匹配结果时，以匹配字符串的首字符作为子串的的第一个字符输出，如果匹配不成功显示没有找到。

本例程的运行结果是：

```
The substring is: national
```

注意：本例程中，匹配成功时的返回结果并不是进行匹配的字符串，而是第一次匹配成功的字符串首字符的指针。

## 12.14 `strtok` 字符串分隔函数

函数原型：char \*strstr(char \*str1, char \*str2);

头文件：#include<string.h>

是否是标准函数：是

函数功能：在字符串中查找单词，这个单词由第二个字符串中定义的分隔符分开，也就是在字符串 `str1` 中查找由字符串 `str2` 定义的分隔符，以分隔符为界，分隔出来的分隔符前面的所有字符组成一个单词，分离出第一个单词后将第一个参数置为空，可以继续分隔第二个单词。

返回值：返回分隔出的单词的指针。



**例程 12-14** 应用 strtok 分隔字符串。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str1="I am very\thappy,to,study\nC\nprogramme";
    char *str2=" ,\t\n";
    char *token;
    printf("%s\n\nTokens:\n",str1);
    token = strtok(str1,str2);
    while( token != NULL )
    {
        printf("%s\n",token);
        token = strtok(NULL,str2);
    }
    getch();
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了三个字符串变量并对前两个变量赋予初值，其中第二个字符串是用于分隔的分隔符集，最后一个 token 用于记录分隔出来的单词。注意第一个字符串中有许多分隔符。

(2) 为了突出分隔结果，首先将字符串 str1 打印出来，它是按照标准输出格式进行输出的。

(3) 程序通过调用 strtok 进行字符串分隔，查找字符串 str1 中首次出现字符串 str2 任意分隔符的位置，然后将分隔符之前的所有字符组成一个单词返回给 token 变量，从而得到分隔出来的首个单词。

(4) 在循环体中，每次循环都要将上一次分隔出来的单词打印出来。将 strtok 函数的第一个参数置为空可以达到继续进行分隔的目的，也就是在上一次分隔出来的单词之后继续进行分隔，直到所有单词都分隔完毕，token 变量会得到空的返回值，结束循环。

(5) 打印分隔结果时，以换行区分每次分隔出的单词。

本例程的运行结果是：

```
I am very      happy,to,study
C
Programme

Tokens:
I
am
very
happy
to
study
C
Programme
```

注意：本例程中，如果在第一次分隔出单词后想继续进行分隔操作，务必要将函数的第一个参数置为空。



## 12.15 strtod 字符串转换成双精度函数

函数原型: `double strtod(char *str, char **endptr);`

头文件: `#include <stdlib.h>`

是否是标准函数: 是

函数功能: 将字符串转换为双精度值, 其中进行转换的字符串必须是双精度数的字符表示格式, 如果字符串中有非法的非数字字符, 则第二个参数将负责获取该非法字符, 即字符串指针 `endptr` 用于进行错误检测, 转换在此非法字符处停止进行。

返回值: 返回转换后的双精度结果。

### 例程 12-15 应用 strtod 将字符串转换为双精度值。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(void)
{
    char str[20], *endptr;
    double result;
    while(1)
    {
        printf("Input a float:");
        gets(str);
        result=strtod(str,&endptr);
        if(result!=1)
            printf("The number is %lf\n",result);
        else
            break;
    }
    getch();
    return 0;
}
```

#### 例程说明:

(1) 首先程序声明了一个用于转换的字符数组和一个用于进行错误检测的字符串指针, 双精度 `result` 用于获取转换结果。

(2) 程序通过循环不断接收从标准输入流中输入的字符串, 并通过调用 `strtod` 将输入的字符串转换为双精度值, 通过返回值获取转换结果, 并通过第二个参数进行错误检测。循环的最后将转换结果打印出来。

(3) 程序规定将字符串“-1”作为循环结束的标志, 除非通过输入结束标志, 否则循环条件总是成立的。

(4) 如果输入一个正确的双精度字符串, 程序会正确地转换并补齐尾数; 如果输入整数, 程序会自动将其转换成双精度数; 如果小数点后面的位数过长, 超过了双精度的范围, 程序会采取截断的方式; 如果输入期间出现了非法字符, 程序停止转换并保留已转换的结果; 如果输入的第一个字符就是非法字符则返回零。

本例程的运行结果是:

```
Input a float: 4.2
```



```
The number is 4.200000
Input a float: 1.2
The number is 1.200000
Input a float: 5.6
The number is 5.600000
Input a float: 34.45abc
The number is 34.450000
Input a float: abc
The number is 0.000000
Input a float: 1
```

注意：本例程中，即便转换出现非法字符循环也不会停止，而只是通过第二个参数捕捉到了非法字符。当然，可以编写程序对非法字符进行处理，本例中并没有这样做，循环以输入结束标志结束。

## 12.16 strtol 字符串转换成长整型函数

函数原型：long strtol(char \*str, char \*\*endptr, int base);

头文件：#include <stdlib.h>

是否是标准函数：是

函数功能：将字符串转换为长整型值，也就是将字符串 str 转换为长整型值，其中进行转换的字符串必须是长整型的字符表示格式。如果字符串中有非法的非数字字符，则第二个参数将负责获取该非法字符，即字符串指针 endptr 用于进行错误检测，转换在此非法字符处停止进行。

返回值：返回转换后的长整型结果。

### 例程 12-16 应用 strtol 将字符串转换为长整型值。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(void)
{
    char str[20], *endptr;
    long result;
    while(1)
    {
        printf("Input a long:");
        gets(str);
        result=strtod(str,&endptr);
        if(result!=-1)
            printf("The number is %ld\n",result);
        else
            break;
    }
    getch();
    return 0;
}
```

#### 例程说明：

(1) 首先程序声明了一个用于转换的字符数组和一个用于进行错误检测的字符串指针，长整型 result 用于获取转换结果。



(2) 程序通过循环不断接收从标准输入流中输入的字符串，并通过调用 `strtol` 将输入的字符串转换为长整型值，通过返回值获取转换结果，并通过第二个参数进行错误检测。循环的最后将转换结果打印出来。

(3) 程序规定将字符串“-1”作为循环结束的标志，除非通过输入结束标志，否则循环条件总是成立的。

(4) 如果输入一个正确的长整型字符串，程序会正确转换；如果输入小数，程序会将小数点后面的截断；如果转换的数过大，超过了长整型范围，程序返回长整型所能接受的最大值；如果输入期间出现了非法字符，程序停止转换并保留已转换的结果；如果输入的第一个字符就是非法字符则返回零。

本例程的运行结果是：

```
Input a long: -15
The number is -15
Input a long: 1234.5678
The number is 1234
Input a long: 333333333333
The number is -1674115755
Input a long: -34abc
The number is -34
Input a long: abc
The number is 0
Input a long: -1
```

注意：本例程中，将字符串中的小数转换为长整型时，程序会将小数点看作非法字符，从而停止转换，因此无论小数点后面的数是多少都会截断，而不是我们习惯上的四舍五入或者五舍六入。

## 12.17 strtoul 字符串转换成无符号长整型函数

函数原型：unsigned long strtoul(char \*str, char \*\*endptr, int base);

头文件：#include<stdlib.h>

是否是标准函数：是

函数功能：将字符串转换为无符号长整型值，也就是将字符串 `str` 转换为无符号长整型值，其中进行转换的字符串必须是无符号长整型的字符表示格式。如果字符串中有非法的非数字字符，则第二个参数将负责获取该非法字符，即字符串指针 `endptr` 用于进行错误检测，转换在此非法字符处停止进行。

返回值：返回转换后的无符号长整型结果。

**例程 12-17** 应用 `strtoul` 将字符串转换为无符号长整型值。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(void)
{
    char str[20], *endptr;
```



```
unsigned long result;
while(1)
{
    printf("Input an unsigned long:");
    gets(str);
    result=strtoul(str,&endptr,0);
    if(result!=100000)
        printf("The number is %lu\n",result);
    else
        break;
}
getch();
return 0;
}
```

#### 例程说明:

(1) 首先程序声明了一个用于转换的字符数组和一个用于进行错误检测的字符串指针, 无符号长整型 `result` 用于获取转换结果。

(2) 程序通过循环不断接收从标准输入流中输入的字符串, 并通过调用 `strtoul` 将输入的字符串转换为无符号长整型值, 通过返回值获取转换结果, 并通过第二个参数进行错误检测。循环的最后将转换结果打印出来。

(3) 程序规定将字符串 "100000" 作为循环结束的标志, 除非通过输入结束标志, 否则循环条件总是成立的。

(4) 如果输入一个正确的长整型字符串, 程序会正确转换; 如果输入一个负数或是非数字字符串, 程序返回零。

本例程的运行结果是:

```
Input a unsigned long: 100
The number is 100
Input a unsigned long: -36
The number is 0
Input a unsigned long:abcdef
The number is 0
Input a unsigned float: 100000
```

注意: 本例程中, 输入负数和字符串 "abcdef" 的时候, 程序会将负号或非数字字符看作非法字符, 从而停止转换并继续运行, 实际上没有发生任何的转换。

## 12.18 strdup 字符串新建拷贝函数

函数原型: `char *strdup(char *str);`

头文件: `#include<stdlib.h>`

是否是标准函数: 是

函数功能: 将字符串复制到新分配的空间位置, 也就是将 `str` 复制到一块新分配的存储空间, 其内部是使用动态分配内存技术实现的, 分配给字符串的空间来自于当前所用内存模式制定的堆。

返回值: 返回指向含有该串拷贝的存储区。



**例程 12-18** 应用 `strdup` 将字符串复制到新建位置处。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *src="This is the buffer text";
    char *dest;
    dest=strdup(src);
    if(!strcmp(src,dest))
        printf("Copy success\n%s\n",dest);
    else
        printf("Copy failure");
    free(dest);
    getch();
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了两个字符串并对第一个字符串赋予初值，此时并未给字符串 `dest` 分配任何空间。

(2) 程序通过调用 `strdup` 将字符串复制到新建位置处，通过动态分配内存技术将新分配一个与字符串 `src` 大小相同的存储区并完成字符串的复制工作，然后返回该存储区并让 `dest` 指向该区域。

(3) 程序通过调用 `strcmp` 比较复制前后的字符串，如果复制成功应当相同，函数返回值为零，并打印拷贝结果。

(4) 由于新分配的存储区是通过动态分配内存技术实现的，因此在程序退出之前要将分配的存储区显示的内容释放。

本例程的运行结果是：

```
Copy success
This is the buffer text
```

注意：本例程中，初学者往往会忽视释放动态分配存储区的操作，表面看起来似乎对程序没有什么影响，但实际上如果不对存储区进行回收会造成内存泄漏，对一些大程序会造成致命的后果。

## 12.19 `strset` 字符串设定函数

函数原型：`char *strset(char *str, char c);`

头文件：`#include <string.h>`

是否是标准函数：否

函数功能：将字符串原有字符全部设定为指定字符，也就是将字符串 `str` 中的所有字符全部用字符 `c` 进行替换。

返回值：返回指向被替换字符串的指针。



**例程 12-19** 应用 `strset` 将字符串设定为指定字符。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[11]="0123456789";
    char symbol='a';
    printf("Before: %s\n",str);
    strset(str,symbol);
    printf("After: %s\n",str);
    getch();
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了一个字符数组和一个字符型变量并赋予初值，字符型变量代表用于替换的字符。

(2) 程序通过调用 `strset` 将原字符串中的所有内容用字符'a'替换，相当于覆盖了原值并重新设定了字符串的值。

(3) 最后将设定前后字符串的值打印出来。

本例程的运行结果是：

```
Before: 0123456789
After: aaaaaaaaaa
```

注意：本例程中，字符数组中只存储了 10 个字符，但是下标却是 11，利用字符串的相关知识可以理解该问题，有兴趣的读者可以将下标改成 10 或在字符串赋值的时候多加一个字符，看看程序的输出结果。

## 12.20 `strrev` 字符串倒转函数

函数原型：char \*strrev(char \*str);

头文件：#include<string.h>

是否是标准函数：否

函数功能：将字符串进行倒转，也就是将字符串 `str` 中的第一个字符与最后一个字符交换，第二个字符与倒数第二个字符交换，……，依此类推。

返回值：返回倒转后字符串的指针。

**例程 12-20** 应用 `strrev` 将字符串倒转。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str = "Able was I ere I saw Elba";
    printf("Before: %s\n",str);
    strrev(str);
    printf("After: %s\n",str);
    getch();
}
```



```
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了一个字符数组并赋予初值，应该注意到字符数组的初值很有意思，类似于回文。

(2) 程序通过调用 `strrev` 将原字符串中的所有内容倒转，第一个字符与最后一个字符交换，第二个字符与倒数第二个字符交换，……，依此类推。

(3) 最后将倒转前后字符串的值打印出来。

本例程的运行结果是：

```
Before:Able was I ere I saw Elba
After:ablE was I ere I saw elbA
```

注意：本例程中，字符数组中的初值并不是严格意义上的回文，将它倒转后会发现与原字符串并不是完全一样。

## 12.21 swab 字符串交换字节函数

函数原型：void swab (char \*from, char \*to, int nbytes);

头文件：#include <string.h>

是否是标准函数：否

函数功能：将字符串的字符进行交换，也就是将字符串 `str` 中的第一个字符与第二个字符交换，第三个字符与第四个字符进行交换，……，依此类推，交换的过程不是在原有的字符串上进行交换操作，而是在目标字符串上保存交换结果，可以通过最后一个参数来限定参与字符交换的位数。

返回值：无返回值。

### 例程 12-21 应用 swab 交换字符串字节。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(void)
{
    char src1[20] = "eHl1 ooWlr d";
    char src2[20] = "eHl1 ooWlrd";
    char dest1[20], dest2[20];
    swab(src1, dest1, strlen(src1));
    dest1[strlen(src1)] = '\0';
    printf("dest: %s\n", dest1);
    swab(dest1, dest1, 2);
    printf("dest: %s\n", dest1);
    swab(src2, dest2, strlen(src2));
    dest2[strlen(src2)] = '\0';
    printf("dest: %s\n", dest2);
    getch();
    return 0;
}
```



**例程说明：**

(1) 首先程序声明了 4 个字符数组，前两个赋予初值表示待交换的字符串，后两个存储交换后的结果。

(2) 程序通过调用 `swab` 使字符串 `str1` 中的所有字节参与交换，交换后一定要补上代表字符串结束的转意说明符 `'\0'`，否则在以后对该字符串进行操作时会产生不可预知的错误。

(3) 程序第二次调用 `swab` 目的是在字符串自身上完成交换，而不是将交换结果保存在新字符串中，因此将源字符串与目标字符串设为相同。这里我们只交换前两字节，由于程序的需要，不要在交换后的两字节后面添加转意说明符 `'\0'`，否则会将后面的字符截断。

(4) 程序第三次调用 `swab` 其结果与我们预想的似乎不尽相同，通过观察打印结果会发现最后一个字符 `'d'` 没有了，这是因为在交换的过程中没有与该字符进行交换的字符，程序的处理就是将该字符删除。

本例程的运行结果是：

```
dest: Hello World
dest: eHllo World
dest: Hello Worl½
```

注意：本例程中，两个源字符串唯一的区别就是最后一个字符 `'d'` 前是否有空格，如果有就发生交换，如果没有就舍去该字符。

## 12.22 strlwr 字符串小写转换函数

函数原型：char \*strlwr(char \*str);

头文件：#include <string.h>

是否是标准函数：否

函数功能：将字符串原有大写字符全部转换为小写字符，也就是将字符串 `str` 中的所有字符变成小写。

返回值：返回指向被转换字符串的指针。

### 例程 12-22 应用 strlwr 将字符串转换成小写字符。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *s="You'll Never Walk Alone";
    printf("%s",strlwr(s));
    getch();
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了一个字符串为待转换字符串并赋予初值。

(2) 程序通过调用 `strlwr` 将字符串中的所有大写字符转换成小写字符，并返回转换后的结果。



(3) 最后将转换结果打印出来。

本例程的运行结果是：

```
you'll never walk alone
```

## 12.23 strupr 字符串大写转换函数

函数原型：char \*strupr(char \*str);

头文件：#include<string.h>

是否是标准函数：否

函数功能：将字符串原有小写字符全部转换为大写字符，也就是将字符串 str 中的所有字符变成大写。

返回值：返回指向被转换字符串的指针。

**例程 12-23** 应用 strupr 将字符串转换成大写字符。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *s=" You'll Never Walk Alone ";
    printf("%s",strupr(s));
    getch();
    return 0;
}
```

例程说明：

(1) 首先程序声明了一个字符串为待转换字符串并赋予初值。

(2) 程序通过调用 strupr 将字符串中的所有小写字符转换成大写字符，并返回转换后的结果。

(3) 最后将转换结果打印出来。

本例程的运行结果是：

```
YOU'LL NEVER WALK ALONE
```

## 12.24 strerror 字符串错误信息函数

函数原型：char \*strerror(int errnum);

头文件：#include<string.h>

是否是标准函数：是

函数功能：获取程序出现错误的字符串信息，也就是根据错误代码 errnum 查找到具体的错误信息。

返回值：返回错误信息。



**例程 12-24** 应用 strerror 查看几种错误信息。

```
#include <stdio.h>
#include <errno.h>
int main(void)
{
    char *error;
    int i;
    for(i=0;i<12;i++)
    {
        error=strerror(i);
        printf("%s",error);
    }
    getch();
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了一个字符串用于获取错误信息，声明的整型变量既作为循环变量又作为错误信息代码。

(2) 程序调用 strerror，根据错误代码获取具体的错误信息，其中代表具体错误信息的字符串在相应的头文件中应给出定义。循环只取了前十二种错误信息，实际的错误种类更多。

(3) 每次循环将具体错误信息打印出来。

本例程的运行结果是：

```
Error 0
Invalid function number
No such file or directory
Path not found
Too many open files
Permission denied
Bad file number
Memory arena trashed
Not enough memory
Invalid memory block address
Invalid environment
Invalid format
```

注意：读者如有兴趣，可以统计系统一共定义了多少种错误信息，通过更改循环变量将各种错误信息打印出来。

## 12.25 atoi 字符串转整型的函数

函数原型：int atoi(const char \*str);

头文件：#include<stdlib.h>

是否是标准函数：是

函数功能：将字符串转换成整型值，也就是将字符串 str 转换成整型值然后获取转换后的结果。

返回值：返回转换后的整型值。

**例程 12-25** 应用 atoi 将字符串转换成整型值。

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *str="12345.67";
    int result;
    result=atoi(str);
    printf("string=%s\ninteger=%d\n",str,result);
    getch();
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了一个字符串作为待转换的字符串，声明的整型变量 result 用于获取转换结果。

(2) 程序通过调用 atoi 将字符串转换为相应的整型变量，然后获取转换结果，转换规则与 strtol 函数相同。

(3) 最后将转换结果打印出来。

本例程的运行结果是：

```
string =12345.67
integer=12345
```

## 12.26 atol 字符串转长整型的函数

函数原型：long atol(const char \*str);

头文件：#include<stdlib.h>

是否是标准函数：是

函数功能：将字符串转换成长整型值，也就是将字符串 str 转换成长整型值，然后获取转换后的结果。

返回值：返回转换后的长整型值。

**例程 12-26** 应用 atol 将字符串转换成长整型值。

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *str="12345.67";
    long result;
    result=atol(str);
    printf("string=%s\nlong=%ld\n",str,result);
    getch();
    return 0;
}
```

**例程说明：**

(1) 首先程序声明了一个字符串作为待转换的字符串，声明的长整型变量 result 用于获取转换结果。



(2) 程序通过调用 `atol` 将字符串转换为相应的长整型变量，然后获取转换结果，转换规则与 `strtoX` 函数相同。

(3) 最后将转换结果打印出来。

本例程的运行结果是：

```
string=12345.67
long =12345
```

## 12.27 atof 字符串转浮点型的函数

函数原型：float `atof`(const char \*str);

头文件：#include <stdlib.h>

是否是标准函数：是

函数功能：将字符串转换成浮点值，也就是将字符串 `str` 转换成浮点值，然后获取转换后的结果。

返回值：返回转换后的浮点值。

### 例程 12-27 应用 `atof` 将字符串转换成浮点值。

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *str="12345.67";
    float result;
    result=atof(str);
    printf("string=%s\nfloat=%f\n",str,result);
    getch();
    return 0;
}
```

例程说明：

(1) 首先程序声明了一个作为待转换的字符串，声明的浮点型变量 `result` 用于获取转换结果。

(2) 程序通过调用 `atof` 将字符串转换为相应的浮点型变量，然后获取转换结果，转换规则与 `strtoX` 函数相同。

(3) 最后将转换结果打印出来。

本例程的运行结果是：

```
string=12345.67
float =12345.669922
```

注意：本例程中，转换成浮点数的结果有些奇怪，它并不等于字符串中变量的值，而是存在一定的误差，虽然误差很小，但是可以看出误差是从原字符串中的最后一位开始的。这是由于在转换过程中，函数内部在实现时采用的转换方式所造成的，如果想避免这种误差，可以使用 `strtoX` 系列函数。

## 12.28 memcpy 字符串拷贝函数

函数原型: `void *memcpy(void *destin, void *source, unsigned n);`

头文件: `#include <string.h>`

是否是标准函数: 是

函数功能: 从 `source` 所指的对象中复制 `n` 个字符到 `destin` 所指的对象中。但是, 如果这种复制发生在重叠对象之间, 其行为是不可预知的。

返回值: `destin`。

### 例程 12-28 利用函数 memcpy 进行字符串拷贝。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *s = "#####";
    char *d = "This is a test for memcpy function";
    char *ptr;
    printf("destination before memcpy: %s\n", d);
    ptr = memcpy(d, s, strlen(s));
    if (ptr)
        printf("destination after memcpy: %s\n", d);
    else
        printf("memcpy failed\n");
    return 0;
}
```

#### 例程说明:

- (1) 首先定义两个字符串 `s` 和 `d` 并赋初值, 且 `d` 的长度大于 `s`。
- (2) 显示字符串 `d` 的原始内容。
- (3) 通过函数 `memcpy` 将字符串 `s` 复制到字符串 `d` 中, 并返回字符串 `d` 的首指针。
- (4) 如果复制成功, 再次显示字符串 `d` 的内容。

本例程的运行结果为:

```
destination before memcpy: This is a test for memcpy function
destination after memcpy: #####test for memcpy function
```

#### 注意:

(1) `memcpy` 与 `strcpy` 的不同在于, 应用 `memcpy` 进行字符串的拷贝可以指定拷贝串的长度。另外 `memcpy` 的参数为 `void` 指针类型, 因此它还可以对非字符型对象进行操作, 而 `strcpy` 只适用于字符串的拷贝。

(2) 如果复制过程发生在重叠对象之间, 其行为是不可预知的。例如下面这个例子:

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char *d = "1234567890";
    char *p;
    p=d+3;
```



```
printf(" %s\n", d);
memcpy(p, d, 6);
printf(" %s\n", d);
return 0;
}
```

由于字符串 p 是字符串 d 的一个子串，在调用 memcpy 时，复制的字符串在 d 和 p 之间重叠，因此该复制行为是不可预知的，结果自然也难以保证。

这段程序的运行结果为：

```
1234567890
1231231230
```

显然，这不是期望得到的结果。

## 12.29 memmove 字块移动函数

函数原型：void \*memmove(void \*destin, void \*source, unsigned n);

头文件：#include <string.h>

是否是标准函数：是

函数功能：从 source 所指的对象中复制 n 个字符到 destin 所指的对象中。与 memcpy 不同的是，当对象重叠时，该函数仍能正确执行。

返回值：destin。

### 例程 12-29 利用函数 memmove 进行字符块的移动。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *s = "#####";
    char *d = "This is a test for memcpy function";
    char *ptr;
    printf("destination before memmove: %s\n", d);
    ptr = memmove(d, s, strlen(s));
    if (ptr)
        printf("destination after memmove: %s\n", d);
    else
        printf("memcpy failed\n");
    return 0;
}
```

#### 例程说明：

- (1) 首先定义两个字符串 s 和 d 并赋初值，且 d 的长度大于 s。
- (2) 显示字符串 d 的原始内容。
- (3) 通过函数 memmove 将字符串 s 复制到字符串 d 中，并返回字符串 d 的首指针。
- (4) 如果复制成功，再次显示字符串 d 的内容。

本例程的运行结果为：

```
destination before memmove: This is a test for memcpy function
destination after memmove: #####test for memcpy function
```

注意：与函数 `memcpy` 不同的是，当对象重叠时，该函数仍能正确执行。例如下面这个例子：

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char *d = "1234567890";
    char *p;
    p=d+3;
    printf(" %s\n", d);
    memmove(p, d, 6);
    printf(" %s\n", d);
    return 0;
}
```

虽然复制的字符串在 `d` 和 `p` 之间重叠，但本段程序的运行结果为：

```
1234567890
1231234560
```

显然，这是期望得到的结果。

这是因为函数 `memmove` 的复制行为，类似于先从 `source` 对象中复制 `n` 个字符到一个与 `source` 和 `destin` 都不重合的含 `n` 个字符的临时数组中作为缓冲，然后从临时数组中再复制 `n` 个字符 `destin` 到所指的對象中。

就本段程序而言，`memmove` 先将字符串 "123456" 复制到一个临时数组中，再将它复制到以 `p` 为首地址的字符串中。

## 12.30 memcmp 字符串比较函数

函数原型：void \*memcmp(char \*s1, char \*s2, unsigned n);

头文件：#include<string.h>

是否是标准函数：是

函数功能：比较 `s1` 所指向的字符串与 `s2` 所指向的字符串的前 `n` 个字符。

返回值：根据 `s1` 所指向的对象与 `s2` 所指向的对象的比较，函数 `memcmp` 分别返回大于、等于、小于 0 的值。

### 例程 12-30 比较两个字符串。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str1="ABCDEF";
    char *str2="ABCDEF";
    int s1,s2;
    s1=memcmp(str1,str2,6);
    s2=memcmp(str1,str2,5);
}
```



```
printf("The comparison of 6 character\n");
if(s1>0)printf("%s>%s\n",str1,str2);
else
    if(s1<0)printf("%s<%s\n",str1,str2);
else
    printf("%s=%s\n",str1,str2);
printf("The comparison of 5 character\n");
if(s2>0)printf("%s>%s\n",str1,str2);
else
    if(s2<0)printf("%s<%s\n",str1,str2);
else
    printf("%s=%s\n",str1,str2);
getchar();
}
```

#### 例程说明:

- (1) 首先初始化两个字符串"ABCDEF"和"ABCDEFf"。
- (2) 然后应用函数 memcmp 将这两个字符串按照不同的字符个数进行比较,将返回的比较结果复制给变量 s1 和 s2。
- (3) 显示比较结果。

本例程的运行结果为:

```
The comparison of 6 character
ABCDEF<ABCDEFf
The comparison of 5 character
ABCDEF=ABCDEFf
```

注意: 由于字符串是从左至右按照字符的 ASCII 码进行比较的,因此在比较 6 个字符时,字符串"ABCDEF"<"ABCDEFf" (f 的 ASCII 值大于 F 的 ASCII 值); 而只比较 5 个字符时,字符串"ABCDEF"="ABCDEFf"。

## 12.31 memchr 字符搜索函数

函数原型: void \*memchr(void \*s, char ch, unsigned n);

头文件: #include<string.h>

是否是标准函数: 是

函数功能: 在数组的前 n 字节中搜索字符 ch。

返回值: 返回一个指针,它指向 ch 在 s 中第一次出现的位置。如果在 s 的前 n 个字符中找不到匹配,返回 NULL。

#### 例程 12-31 应用函数 memchr 搜索一个字符串的子串。

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char *str="I love China\n";
    char *p;
    p=memchr(str,'C',strlen(str));
    if(p)
```

```
    printf("%s",p);
else
    printf("The character was not found\n");
getchar();
}
```

例程说明:

- (1) 首先初始化字符串"I love China\n", 将首地址赋值给 str。
- (2) 在字符串 str 中查找字符'C'出现的位置, 并返回以第一个字符'C'开头的字符串的指针。
- (3) 如果返回值不为 NULL 则打印该子串。

本例程的运行结果为:

```
China
```

## 12.32 memset 字符加载函数

函数原型: void \*memset(void \*s, int c, unsigned n);

头文件: #include <string.h>

是否是标准函数: 是

函数功能: 把 c 复制到 s 所指对象的前 n 个字符的每一个字符中。

返回值: s 的值。

**例程 12-32** 应用 memset 函数替换字符串中的字符。

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char *str="AAAAAAAAAAAAAAAAAAAA";
    printf("The original string is: %s\n",str);
    memset(str,'B',9);
    printf("The string after memset is:%s\n",str);
}
```

例程说明:

- (1) 首先初始化字符串"AAAAAAAAAAAAAAAAAAAA", 并将首地址赋值给 str。
- (2) 显示该字符串。
- (3) 利用函数 memset 将字符串 str 的前 9 个字符替换为'B'。
- (4) 显示替换后的字符串。

本例程的运行结果为:

```
The original string is:  AAAAAAAAAAAAAAAAAAAAA
The string after memset is:BBBBBBBBBAAAAAAAAAAAA
```



C 语言库函数中有强大的数学函数, 用来支持复杂的数学运算。这些数学函数主要定义在 `math.h` 头文件中, 当然其他头文件中也定义有支持数学运算的函数。本章将介绍这些数学函数。

### 13.1 abs、labs、fabs 求绝对值函数

函数原型: `int abs(int x);`

`long labs(long x);`

`double fabs(double x);`

头文件: `#include <math.h>`

是否是标准函数: 是

函数功能: 函数 `int abs(int x)` 是求整数 `x` 的绝对值; 函数 `long labs(long n)` 是求长整型数 `x` 的绝对值; 函数 `double fabs(double x)` 是求双精度浮点型数 `x` 的绝对值。

返回值: 返回计算结果。

#### 例程 13-1 计算整数的绝对值。

```
#include <math.h>
int main(void)
{
    int x = -56;
    printf("number: %d absolute value: %d\n", x, abs(x));
    return 0;
}
```

例程说明:

本例程通过 `abs` 函数计算出整型数 `-56` 的绝对值 `56`, 并在屏幕上显示结果。

本例程的运行结果是:

```
number: -56 absolute value: 56
```

#### 例程 13-2 计算长整型数的绝对值。

```
#include <math.h>
int main(void)
{
    long x = -12345678L;
    printf("number: %ld absolute value: %ld\n", x, labs(x));
    return 0;
}
```

**例程说明：**

本例程通过 labs 函数计算出长整型数-12 345 678 的绝对值 12 345 678，并在屏幕上显示结果。

本例程的运行结果是：

```
number: -12345678 absolute value: 12345678
```

**例程 13-3 计算浮点型数的绝对值。**

```
#include <math.h>
int main(void)
{
    float x = -128.0;
    printf("number: %f absolute value: %f\n", x, fabs(x));
    return 0;
}
```

**例程说明：**

本例程通过 fabs 函数计算出浮点数-128.0 的绝对值 128.0，并在屏幕上显示结果。

本例程的运行结果是：

```
number: -128.000000 absolute value: 128.000000
```

## 13.2 div、ldiv 除法函数

函数原型：div\_t div(int number, int denom);

ldiv\_t ldiv(long lnumber, long ldenom);

头文件：#include<stdlib.h>

是否是标准函数：是

函数功能：函数 div 是将两个整数 number 和 denom 相除，返回商和余数；函数 ldiv 是将两个长整数 lnumber 和 ldenom 相除，返回商和余数。

返回值：函数 div 返回 div\_t 类型的值；函数 ldiv 返回 ldiv\_t 类型的值。

**例程 13-4 两整数相除，求其商和余数。**

```
#include <stdlib.h>
#include <stdio.h>
div_t x;
int main(void)
{
    x = div(11,5);
    printf("11 div 5 = %d remainder %d\n", x.quot, x.rem);
    return 0;
}
```

**例程说明：**

本例程通过 div 函数将 11 和 5 相除，返回其商和余数。

注意：div 函数并不是<math.h>中的函数，而是<stdlib.h>中的函数。<stdlib.h>中包含存储分配函数和其他一些函数。但由于 div 函数具有数学计算的功能，因此将其归类到数学函数中。



`div_t` 是 `<stdlib.h>` 中定义的数据类型，它是一个结构体，定义如下：

```
typedef struct
{
    int quot;      /*商*/
    int rem;       /*余数*/
} div_t;
```

其中包含两个域：商和余数。`div` 函数将两个整数相除，返回一个 `div_t` 类型的值。该函数的运行结果是：

```
11 div 5 = 2 remainder 1
```

### 例程 13-5 两长整数相除，求其商和余数。

```
#include <stdlib.h>
#include <stdio.h>
ldiv_t lx;
int main(void)
{
    lx = ldiv(200000L, 70000L);
    printf("200000 div 70000 = %ld remainder %ld\n", lx.quot, lx.rem);
    return 0;
}
```

#### 例程说明：

本例程通过 `ldiv` 函数将长整数 200 000 与 70 000 相除，并返回其商和余数。

注意：同函数 `div` 一样，函数 `ldiv` 是 `<stdlib.h>` 中的函数。

`ldiv_t` 是 `<stdlib.h>` 中定义的数据类型，它是一个结构体，定义如下：

```
typedef struct {
    long    quot;
    long    rem;
} ldiv_t;
```

其中包含两个域：商和余数。`ldiv` 函数将两个长整数相除，返回一个 `ldiv_t` 类型的值。该函数的运行结果是：

```
200000 div 70000 = 2 remainder 60000
```

## 13.3 ceil 向上舍入函数

函数原型：double `ceil`(double `x`);

头文件：#include <math.h>

是否是标准函数：是

函数功能：将双精度数 `x` 向上舍入，即取它的最大整数。例如：`ceil(123.400000)=124.000000`。

返回值：返回计算结果。

**例程 13-6 数值的向上舍入。**

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double num = 123.400000;
    double up;
    up = ceil(num);
    printf("The original number    %lf\n", num);
    printf("The num rounded up    %lf\n", up);
    return 0;
}
```

**例程说明：**

本例程通过函数 `ceil` 将双精度数 123.400000 向上舍入，得到的结果为 124.000000，并在屏幕上显示运算结果。本例程的运行结果是：

```
The original number    123.400000
The num rounded up    124.000000
```

## 13.4 floor 向下舍入函数

函数原型：double floor(double x);

头文件：#include<math.h>

是否是标准函数：是

函数功能：将双精度数 `x` 向下舍入，即取它的最小整数。例如：`floor(123.400000)=123.000000`。

返回值：返回计算结果。

**例程 13-7 数值的向下舍入。**

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double num = 123.400000;
    double up;
    up = floor(num);
    printf("The original number    %lf\n", num);
    printf("The num rounded down    %lf\n", up);
    return 0;
}
```

**例程说明：**

本例程通过函数 `floor` 将双精度数 123.400000 向下舍入，得到的结果为 123.000000，并在屏幕上显示运算结果。

本例程的运行结果是：

```
The original number    123.400000
The num rounded down    123.000000
```



## 13.5 fmod 求模函数

函数原型: `double fmod(double x, double y);`

头文件: `#include <math.h>`

是否是标准函数: 是

函数功能: 计算  $x$  对  $y$  的模, 即  $x/y$  的余数。

返回值: 返回计算结果, 即余数的双精度。

### 例程 13-8 计算两数的余数。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    float x,y;
    x=12.580000;
    y=2.600000;
    printf("12.580000/2.600000: %f\n",fmod(x,y));
    getchar();
    return 0;
}
```

例程说明:

本例程通过函数 `fmod` 求双精度数 12.580000 和 2.600000 的模, 其结果为: 2.180000。

本例程的运行结果是:

```
12.580000/2.600000: 2.180000
```

## 13.6 frexp 分解浮点数函数

函数原型: `double frexp(double val, int *exp);`

头文件: `#include <math.h>`

是否是标准函数: 是

函数功能: 把浮点数或双精度数  $val$  分解为数字部分 (尾数部分)  $x$  和以 2 为底的指数部分  $n$ 。即  $val=x*2^n$ , 其中  $n$  存放在 `exp` 指向的变量中。

返回值: 返回尾数部分  $x$  的双精度值, 且  $0.5 \leq x < 1$ 。

### 例程 13-9 应用函数 `frexp` 分解浮点数。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    float x;
    int exp;
    x=frexp(64.0,&exp);
    printf("64.0=%f*2^%d",x,exp);
    getchar();
    return 0;
}
```

**例程说明：**

本例程通过函数 `frexp` 将浮点数 64.0 分解为尾数 0.5 和以 2 为底的指数 7。该函数将指数 7 存放在变量 `exp` 中，并返回一个双精度的尾数 0.500000。

本例程的运行结果是：

```
64.0=0.50*2^7
```

## 13.7 ldexp 装载浮点数函数

函数原型： `double ldexp(double val, int exp);`

头文件： `#include <math.h>`

是否是标准函数： 是

函数功能： 功能与函数 `frexp` 相反，它将给定的尾数、指数装载成相应的双精度数或浮点数。即计算  $val \times 2^n$ ，其中  $n$  为参数 `exp` 的值。

返回值： 返回  $val \times 2^n$  的计算结果。

### 例程 13-10 应用函数 `ldexp` 装载浮点数。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double value;
    double x = 3.000000;
    value = ldexp(x,3);
    printf("The ldexp value is: %lf\n", value);
    getchar();
    return 0;
}
```

**例程说明：**

本例程通过函数 `ldexp` 将尾数 3.000000 与指数 3 装载成相应的双精度数。即： $3.000000 \times 2^3 = 24.000000$ ，该函数返回一个双精度数。

本例程的运行结果是：

```
The ldexp value is: 24.000000
```

## 13.8 modf 分解双精度数函数

函数原型： `double modf(double num, double *i);`

头文件： `#include <math.h>`

是否是标准函数： 是

函数功能： 把双精度数 `num` 分解为整数部分和小数部分，并把整数部分存到 `i` 指向的单元中。

返回值： 返回 `num` 的小数部分的双精度值。



**例程 13-11** 应用函数 `modf` 分解双精度数。

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double fraction, integer;
    double number = 12345.6789;
    fraction = modf(number, &integer);
    printf("The integer and the fraction of %lf are %lf and %lf\n",
        number, integer, fraction);
    return 0;
}
```

**例程说明：**

本例程将双精度数 12 345.6789 分解为整数部分和小数部分，并将整数部分存入变量 `integer` 中，返回小数部分，最后在屏幕上显示结果。

本例程的运行结果是：

```
The integer and the fraction of 12345.678900 are 12345.000000 and 0.678900
```

## 13.9 `exp` 求 $e$ 的 $x$ 次幂函数

函数原型： `double exp(double x);`

头文件： `#include <math.h>`

是否是标准函数： 是

函数功能： 计算自然常数  $e$  的  $x$  幂。

返回值： 返回计算结果的双精度值。

**例程 13-12** 计算  $e^x$ （说明： $e=2.718281828\dots$ ）。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result;
    double x = 3.0;
    result = exp(x);
    printf("'e' raised to the power of %lf (e ^ %lf) = %lf\n", x, x, result);
    return 0;
}
```

**例程说明：**

本例程应用函数 `exp` 计算  $e^3$ ，该函数返回计算结果的双精度值。

本例程的运行结果是：

```
'e' raised to the power of 3.000000 (e ^ 3.000000) = 20.085537
```

## 13.10 `log`、`log10` 对数函数

函数原型： `double log(double x);`

double log10(double x);

头文件: #include<math.h>

是否是标准函数: 是

函数功能: 求对数。函数 log 是求以 e 为底的 x 的对数 (自然对数), 即  $\ln x$ ; 函数 log10 是求以 10 为底的 x 的对数, 即  $\log_{10} x$ 。

返回值: 返回计算结果的双精度值。

### 例程 13-13 计算 $\ln x$ 。

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double result;
    double x = M_E;
    result = log(x);
    printf("The natural log of %lf is %lf\n", x, result);
    return 0;
}
```

例程说明:

本例程应用函数 log 计算双精度数 M\_E 的自然对数, 其中 M\_E 为<math.h>中定义的常数, #define M\_E 2.71828182845904523536 就等于 e。

本例程的运行结果是:

```
The natural log of 2.718282 is 1.000000
```

### 例程 13-14 计算 $\log_{10} x$ 。

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double result;
    double x = 1000.0 ;
    result = log10(x);
    printf("The common log of %lf is %lf\n", x, result);
    return 0;
}
```

例程说明:

本例程应用函数 log10 计算双精度数 1000.0 的以 10 为底的对数, 该函数返回的结果仍是双精度数。

本例程的运行结果是:

```
The common log of 1000.000000 is 3.000000
```

## 13.11 hypot 求直角三角形斜边长函数

函数原型: double hypot(double x, double y);

头文件: #include<math.h>



是否是标准函数：是

函数功能：x, y 为给定的直角三角形两直角边，求该直角三角形的斜边。

返回值：返回计算结果的双精度值。

#### 例程 13-15 根据两直角边求斜边的长。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result;
    double x = 3.0;
    double y = 4.0;
    result = hypot(x, y);
    printf("The hypotenuse is: %lf\n", result);
    return 0;
}
```

例程说明：

本例程中，已知两直角边长度：x = 3.0; y = 4.0，应用函数 hypot 求出其斜边长度。

本例程的运行结果是：

```
The hypotenuse is: 5.000000
```

## 13.12 pow、pow10 指数函数

函数原型：double pow(double x, double y);

double pow10(int x);

头文件：#include<math.h>

是否是标准函数：是

函数功能：指数函数。函数 pow 是求 x 的 y 次方；函数 pow10 相当于 pow(10.0,x)，是求 10 的 x 次方。

返回值：返回计算结果的双精度值。

#### 例程 13-16 计算 $x^y$ 。

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double x = 2.0, y = 10.0;
    printf("The result of %lf raised to %lf is %lf\n", x, y, pow(x, y));
    return 0;
}
```

例程说明：

本例程中，应用函数 pow 计算  $2^{10}$ ，并将结果的双精度值返回。

本例程的运行结果是：

```
The result of 2.000000 raised to 10.000000 is 1024.000000
```

**例程 13-17** 计算  $10^x$ 。

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double x = 2.0;
    printf("The result of 10 raised to %lf is %lf\n", x, pow10(x));
    return 0;
}
```

**例程说明：**

本例程中，应用函数 `pow10` 计算  $10^2$ ，并将结果的双精度值返回。

本例程的运行结果是：

```
The result of 10 raised to 2.000000 is 100.000000
```

## 13.13 sqrt 开平方函数

函数原型：double sqrt(double x);

头文件：#include<math.h>

是否是标准函数：是

函数功能：求双精度数  $x$  ( $x \geq 0$ ) 的算术平方根。

返回值：返回计算结果的双精度值。

**例程 13-18** 计算双精度数的平方根。

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double result, x = 4.0;
    result = sqrt(x);
    printf("The square root of %lf is %lf\n", x, result);
    return 0;
}
```

**例程说明：**

本例程中，应用函数 `sqrt` 计算出 4.0 的平方根，并将结果的双精度值返回。

本例程的运行结果是：

```
The square root of 4.000000 is 2.000000
```

## 13.14 rand 产生随机整数函数

函数原型：int rand(void);

头文件：#include<math.h>

是否是标准函数：是

函数功能：产生-90~32 767 之间的随机整数。



返回值：产生的随机整数。

#### 例程 13-19 利用函数 rand 产生处于 0~99 之间的 5 个随机整数。

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    int i;
    printf("Random numbers from 0 to 99\n");
    for(i=0; i<5; i++)
        printf("%d ", rand() % 100);
    return 0;
}
```

##### 例程说明：

循环地应用函数 rand 产生随机整数，并利用 rand() % 100 将范围控制在 0~99 之间。共产生 5 个 0~99 之间的随机整数。

本例程的运行结果是：

```
Random numbers from 0 to 99
46 30 82 90 56
```

注意：rand 不是<math.h>中定义的函数，而是<stdlib.h>中定义的函数。因此要在源程序中包含<stdlib.h>头文件。

## 13.15 srand 设置随机时间的种子函数

函数原型：int srand (unsigned int seed);

头文件：#include<math.h>

是否是标准函数：是

函数功能：设置随机时间的种子，常与 rand() 结合使用。如果直接用 rand 函数产生随机数，每次运行程序的结果都相同。

返回值：无。

#### 例程 13-20 产生不同的随机整数序列。

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main(void)
{
    int i;
    time_t t;
    srand((unsigned) time(&t));
    printf("Random numbers from 0 to 99\n");
    for(i=0; i<5; i++)
        printf("%d ", rand() % 100);
    return 0;
}
```



**例程说明：**

(1) 首先程序应用函数 `time` 获取系统时间作为种子，并强制转换为 `unsigned` 型变量，作为函数 `srand` 的参数。

(2) 再利用 `rand` 函数产生 5 个 0~99 之间的随机整数。这样每次运行该程序都以当时的系统时间作为随机时间的种子，产生的随机数就不会重复了。

注意：`time_t` 是 `<time.h>` 中定义的数据类型，用以描述时间；而函数 `time` 可以获取当前的系统时间。运行两次本例程，可得到两组不同的随机数序列：

```
Random numbers from 0 to 99
23 16 92 26 99
Random numbers from 0 to 99
60 72 77 40 45
```

## 13.16 sin 正弦函数

函数原型：`double sin(double x);`

头文件：`#include<math.h>`

是否是标准函数：是

函数功能：求  $x$  的正弦值，这里  $x$  为弧度。

返回值：计算结果的双精度值。

**例程 13-21 求  $\sin x$ 。**

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    float x;
    x=M_PI/2;
    printf("sin(PI/2)=%f",sin(x));
    getchar();
    return 0;
}
```

**例程说明：**

本例程应用 `sin` 函数计算  $\pi/2$  的正弦值，即  $\sin(\pi/2)$ 。返回计算结果的双精度值。

注意：`M_PI` 是 `<math.h>` 中定义的  $\pi$  值常量。本例程的运行结果是：

```
sin(PI/2)=1.00000
```

## 13.17 asin 反正弦函数

函数原型：`double asin(double x);`

头文件：`#include<math.h>`



是否是标准函数：是

函数功能：求  $x$  的反正弦值，这里  $x$  为弧度， $x$  的定义域为  $[-1.0, 1.0]$ ， $\arcsin x$  值域为  $[-\pi/2, +\pi/2]$ 。

返回值：计算结果的双精度值。

#### 例程 13-22 求 $\arcsin x$ 。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result;
    double x = 1.0;
    result = asin(x);
    printf("arcsin %lf is %lf\n", x, result);
    return(0);
}
```

例程说明：

本例程应用函数  $\text{asin}$  计算 1.0 的反正弦值，即  $\arcsin 1$ ，再返回计算结果的双精度值。本例程的运行结果是：

```
arcsin 1.000000 is 1.570796
```

## 13.18 cos 余弦函数

函数原型：double cos(double x);

头文件：#include<math.h>

是否是标准函数：是

函数功能：求  $x$  的余弦值，这里  $x$  为弧度。

返回值：计算结果的双精度值。

#### 例程 13-23 求 $\cos x$ 。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result;
    double x = M_PI;
    result = cos(x);
    printf("cos(PI) is %lf\n", result);
    return 0;
}
```

例程说明：

本例程应用  $\cos$  函数计算  $\pi$  的余弦值，即  $\cos \pi$ ，再返回计算结果的双精度值。本例程的运行结果是：

```
cos(PI) is -1.000000
```



## 13.19 acos 反余弦函数

函数原型: `double acos(double x);`

头文件: `#include <math.h>`

是否是标准函数: 是

函数功能: 求  $x$  的反余弦值, 这里  $x$  为弧度,  $x$  的定义域为  $[-1.0, 1.0]$ ,  $\arccos x$  的值为  $[0, \pi]$ 。

返回值: 计算结果的双精度值。

### 例程 13-24 求 $\arccos x$ 。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result;
    double x = 1.0;
    result = acos(x);
    printf("arccos %lf=%lf\n", x, result);
    return 0;
}
```

例程说明:

本例程应用函数 `acos` 计算 1.0 的反余弦值, 即  $\arccos 1$ , 然后返回计算结果的双精度值。本例程的运行结果是:

```
arccos 1.000000=0.000000
```

## 13.20 tan 正切函数

函数原型: `double tan(double x);`

头文件: `#include <math.h>`

是否是标准函数: 是

函数功能: 求  $x$  的正切值, 这里  $x$  为弧度, 其中  $x \neq k(\pi/2)$ ,  $k$  为整数。

返回值: 计算结果的双精度值。

### 例程 13-25 求 $\tan x$ 。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result, x;
    x = M_PI/4;
    result = tan(x);
    printf("tan (PI/4)=%lf\n", result);
    return 0;
}
```



**例程说明:**

本例程应用 `tan` 函数计算  $\pi/4$  的正切值, 即  $\tan(\pi/4)$ , 并返回计算结果的双精度值。  
本例程的运行结果是:

```
tan (PI/4)=1.00000
```

## 13.21 atan 反正切函数

函数原型: `double atan(double x);`

头文件: `#include<math.h>`

是否是标准函数: 是

函数功能: 求  $x$  的反正切值, 这里  $x$  为弧度,  $x$  的定义域为  $(-\infty, +\infty)$ ,  $\arctan x$  的值为  $(-\pi/2, +\pi/2)$ 。

返回值: 计算结果的双精度值。

**例程 13-26 求  $\arctan x$ 。**

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result;
    double x = 1.0;
    result = atan(x);
    printf("arctan %lf = %lf\n", x, result);
    return(0);
}
```

**例程说明:**

本例程应用函数 `atan` 计算 1.0 的反正切值, 即  $\arctan 1$ , 并返回计算结果的双精度值。  
本例程的运行结果是:

```
arctan 1.000000 = 0.785398
```

## 13.22 atan2 反正切函数

函数原型: `double atan2(double y, double x);`

头文件: `#include<math.h>`

是否是标准函数: 是

函数功能: 求  $y/x$  的反正切值。

返回值: 计算结果的双精度值。

**例程 13-27 求  $\arctan(y/x)$ 。**

```
#include <stdio.h>
#include <math.h>
int main(void)
{
```



```
double result;  
double x = 10.0, y = 5.0;  
result = atan2(y, x);  
printf("arctan %lf = %lf\n", (y / x), result);  
return 0;  
}
```

**例程说明：**

本例程应用函数 `atan2` 计算 `10.0/5.0` 的反正切值，即 `arctan0.5`，并返回计算结果的双精度值。

本例程的运行结果是：

```
arctan 0.500000 = 0.463648
```

## 13.23 sinh 双曲正弦函数

函数原型：`double sinh(double x);`

头文件：`#include <math.h>`

是否是标准函数：是

函数功能：计算  $x$  的双曲正弦值，其中  $sh(x)=(e^x-e^{-x})/2$ 。

返回值：计算结果的双精度值。

**例程 13-28 求  $x$  的双曲正弦值  $sh(x)$ 。**

```
#include <stdio.h>  
#include <math.h>  
int main(void)  
{  
    double result, x = 0.5;  
    result = sinh(x);  
    printf("sh(%lf) = %lf\n", x, result);  
    return 0;  
}
```

**例程说明：**

本例程应用函数 `sinh` 计算 `0.5` 的双曲正弦值，即 `sh(0.5)`，并返回计算结果的双精度值。  
本例程的运行结果是：

```
sh(0.500000) = 0.521095
```

## 13.24 cosh 双曲余弦函数

函数原型：`double cosh(double x);`

头文件：`#include <math.h>`

是否是标准函数：是

函数功能：计算  $x$  的双曲余弦值，其中  $ch(x)=(e^x+e^{-x})/2$ 。

返回值：计算结果的双精度值。



**例程 13-29** 求  $x$  的双曲余弦值  $\text{ch}(x)$ 。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result;
    double x = 0.5;
    result = cosh(x);
    printf("ch(%lf) = %lf\n", x, result);
    return 0;
}
```

**例程说明：**

本例程应用函数 `cosh` 计算 0.5 的双曲余弦值，即  $\text{ch}(0.5)$ ，并返回计算结果的双精度值。本例程的运行结果是：

```
ch(0.500000) = 1.127626
```

## 13.25 tanh 双曲正切函数

函数原型：double `tanh`(double  $x$ );

头文件：#include <math.h>

是否是标准函数：是

函数功能：求  $x$  的双曲正切值，其中  $\text{th}(x) = \text{sh}(x)/\text{ch}(x) = (e^x - e^{-x}) / (e^x + e^{-x})$ 。

返回值：计算结果的双精度值。

**例程 13-30** 求  $x$  的双曲正切值  $\text{th}(x)$ 。

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result, x;
    x = 0.5;
    result = tanh(x);
    printf("th(%lf) = %lf\n", x, result);
    return 0;
}
```

**例程说明：**

本例程应用函数 `tanh` 计算 0.5 的双曲正切值，即  $\text{th}(0.5)$ ，并返回计算结果的双精度值。本例程的运行结果是：

```
th(0.500000) = 0.462117
```



时间函数主要定义在头文件<time.h>中，此类函数的功能是获得系统时间或者对得到的时间进行格式转换等。本章将介绍这些时间函数。

## 14.1 clock 测定运行时间函数

函数原型: `clock_t clock(void);`

头文件: `#include<time.h>`

是否是标准函数: 是

函数功能: 确定所用的处理器时间。函数 `clock` 返回实现环境中从程序运行开始所用的处理器时间的最佳近似值，仅与程序启动有关。`clock` 函数无参数。

返回值: 如果成功，返回从程序开始运行经过的时间；否则（系统没有内部时钟）返回-1。

### 例程 14-1 应用 clock 函数计算程序运行时间。

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
int main(void)
{
    clock_t start, end;
    /*程序运行到现在的时间*/
    start = clock();
    /*间隔 1 秒*/
    sleep(1);
    /*程序运行到现在的时间*/
    end = clock();
    printf("The time was: %f\n", (end - start) / CLK_TCK);
    return 0;
}
```

#### 例程说明:

(1) 首先用 `clock` 函数记录下程序运行到当前所用的时间，并将该时间存入 `clock_t` 类型的变量 `start` 中。

(2) 应用 `sleep` 函数使程序暂停 1 秒。

(3) 再用 `clock` 函数记录下程序运行到当前所用的时间，并将该时间存入 `clock_t` 类型的变量 `end` 中。

(4) 计算出时间差 (`end-start`) 获得程序暂停 `sleep` 的时间。再除以常量 `CLK_TCK`，转化为以秒为单位。



本例程的执行结果为：

```
The time was:0.989011
```

注意：CLK\_TCK 是系统常量。

## 14.2 difftime 计算时间差函数

函数原型：double difftime(time\_t time2, time\_t time1);

头文件：#include<time.h>

是否是标准函数：是

函数功能：计算两个日历时间 time1 和 time2 的时间间隔。其中 time1 为指定的第一个时间，time2 为指定的第二个时间，time1 要小于或等于 time2。

返回值：以秒为单位的 double 类型时间差。

### 例程 14-2 应用函数 difftime 计算时间差。

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
#include <conio.h>
int main(void)
{
    time_t first, second;
    clrscr();
    /*获得系统时间*/
    first = time(NULL);
    /*等待 2 秒*/
    sleep(2);
    /*再次获得系统时间*/
    second = time(NULL);
    printf("The Interval is: %f seconds\n",difftime(second,first));
    getch();
    return 0;
}
```

例程说明：

- (1) 首先通过 time 函数获得系统时间，并将其存储在 time\_t 类型变量 first 中。
- (2) 应用 sleep 函数使程序暂停 2 秒。
- (3) 再次通过 time 函数获得系统时间，并将其存储在 time\_t 类型变量 second 中。
- (4) 通过函数 difftime 获得 first 和 second 的时间间隔，并显示在屏幕上。

本例程的执行结果为：

```
The Interval is: 2.000000 seconds
```

## 14.3 mktime 时间类型转换函数

函数原型：time\_t mktime(struct tm\*timeptr);



头文件: #include<time.h>

是否是标准函数: 是

函数功能: 将 tm 类型的结构指针 timeptr 指向的结构体中的日期与时间转换为 time\_t 类型的日期和时间, 并返回。

返回值: time\_t 类型的日期和时间, 如果日历不能被表达, 则返回-1。

### 例程 14-3 输出指定日期是一周的哪一天。

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
#include <conio.h>
int main(void)
{
    char *week_day [8]={ "Sun",
                          "Mon",
                          "Tue",
                          "Wed",
                          "Fri",
                          "Sat",
                          "Unknow",
                          };

    struct tm t;
    /*指定日期时间*/
    t.tm_year=99;
    t.tm_mon=1;
    t.tm_mday=1;
    t.tm_hour=0;
    t.tm_min=0;
    t.tm_sec=1;
    t.tm_isdst=-1;
    /*调用函数 mktime 设置 tm_wday 成员*/
    if(mktime(&t)==-1)
        t.tm_wday=7;
    printf("The day is:%s",week_day[t.tm_wday]);
    getch();
    return 0;
}
```

#### 例程说明:

- (1) 首先定义一个 struct tm 结构类型的变量 t, 并指定 t 的日期与时间。
  - (2) 通过函数 mktime 将 t 中指定的分段时间转换为日历时间。如果日历时间不能被表达, 则返回-1, 将 t.tm\_wday 赋值为 0。
  - (3) 根据字符串数组 week\_day 的初值, 显示 t 中指定的时间是一周中的哪一天。
- 本例程的运行结果为:

```
The day is:Mon
```

提示: mktime 这个函数是一个 C98 的标准函数, 所以列在这里, 但是在 TC 的 time.h 库文件中并没有定义该函数, 所以该函数无法在 TC 环境下运行。推荐使用 C 程度的开发环境 lcc win32, 在该环境下可以运行 mktime 函数。

#### 注意:

- (1) struct tm 是 time.h 中定义的结构体, 用来指定分段日期与时间。tm 结构如下:



```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

(2) 函数 `mktime` 的作用是将 `tm` 类型结构指针指定的日期和时间转换为 `time_t` 类型的日历时间, 该值与函数 `time` 返回值的编码方式相同。函数 `mktime` 调用成功时, 成员 `tm_wday` 被设置为适当的值, 并返回一个 `time_t` 类型的日历时间。因此本例程中, 如果 `mktime(&t) == -1`, 则表示函数 `mktime` 执行不成功, 将 `t.tm_wday` 置为 7, 输出 "Unknow"; 否则 `t.tm_wday` 会被设置为适当的值。

## 14.4 time 获取系统时间函数

函数原型: `time_t time(time_t *tp);`

头文件: `#include <time.h>`

是否是标准函数: 是

函数功能: 获取系统时间。

返回值: `time_t` 类型的当前日历时间的最佳近似值, 如果日历不能被表达, 则返回 -1。

### 例程 14-4 应用函数 time 获取系统时间。

```
#include <time.h>
#include <stdio.h>
int main(void)
{
    time_t t;
    t = time(&t);
    printf("The number of seconds since January 1, 1970 is %ld", t);
    return 0;
}
```

例程说明:

- (1) 首先定义 `time_t` 类型的变量 `t`。
- (2) 通过函数 `time` 获取从 GMT1970 年 1 月 1 日 00:00:00 开始经过的秒数。
- (3) 输出该秒数。

本例程的运行结果为:

```
The number of seconds since January 1, 1970 is 1047682192
```



## 14.5 asctime 日期和时间转换函数

函数原型: `char *asctime(const struct tm *tblock);`

头文件: `#include <time.h>`

是否是标准函数: 是

函数功能: 本函数把指定的 `tm` 结构类型的日期(分段日期)转换成下列格式的字符串:

`Mon Nov 21 11:31:54 1983\n\0`。

返回值: 转换后的字符串指针。

### 例程 14-5 用 asctime 函数转换时间格式。

```
#include <stdio.h>
#include <string.h>
#include <time.h>
int main(void)
{
    struct tm t;
    char str[80];
    /*设置 tm 结构类型变量 t 的时间成员 */
    t.tm_sec    = 1;        /* 秒 */
    t.tm_min    = 30;       /* 分钟 */
    t.tm_hour   = 9;        /* 时 */
    t.tm_mday   = 22;       /* 日 */
    t.tm_mon    = 11;       /* 月 */
    t.tm_year   = 56;       /* 年 */
    t.tm_wday   = 4;        /* 星期 */
    t.tm_yday   = 0;        /* 不必设置 */
    t.tm_isdst  = 0;        /* 不必设置 */
    /*格式转换 */
    strcpy(str, asctime(&t));
    printf("%s\n", str);
    return 0;
}
```

例程说明:

- (1) 首先定义 `tm` 结构类型的变量 `t`, 并设置 `t` 的时间成员。
- (2) 通过函数 `asctime` 将 `t` 的时间转换为指定格式。
- (3) 输出转换后指定格式的字符串。

本例程的运行结果为:

```
Thu Dec 22 09:30:01 1956
```

注意: 函数 `asctime` 返回指向转换后的字符串指针。本例程通过函数 `strcpy` 将指定格式的字符串的指针复制给 `str`, 并通过 “%s” 格式符输出该指定格式的字符串。

## 14.6 ctime 时间转换函数

函数原型: `char *ctime(const time_t *time);`

头文件: `#include <time.h>`



是否是标准函数：是

函数功能：将 time 所指向的日历时间转换为字符串形式的本地时间。它等价于函数调用 asctime(localtime(timer))。字符串的格式为：DDD MMM dd hh:mm:ss YYYY。

返回值：转换后的字符串指针。

#### 例程 14-6 用 ctime 函数转换时间格式。

```
#include <stdio.h>
#include <time.h>
int main(void)
{
    time_t t;
    time(&t);
    printf("Today's date and time: %s\n", ctime(&t));
    return 0;
}
```

例程说明：

- (1) 首先定义 time\_t 类型的变量 t。
- (2) 应用函数 time 获取系统时间。
- (3) 通过函数 ctime 将获取的日历时间 time\_t 转换为规定格式的字符串表示。

本例程的运行结果为：

```
Today's date and time: Sat Nov 10 00:57:14 2007
```

注意：函数 ctime 是将日历时间直接转换为规定格式的字符串表示：DDD MMM dd hh:mm:ss YYYY，其中，“DDD”表示一星期中的某一天，例如“Sat”表示星期六；“MMM”表示月份，例如“Nov”表示十一月；dd hh:mm:ss 为时钟显示；YYYY 为年份。

## 14.7 gmtime 将日历时间转换为 GMT

函数原型：struct tm \*gmtime(const time\_t \*timer);

头文件：#include<time.h>

是否是标准函数：是

函数功能：把日期和时间转换为格林尼治标准时间（GMT）。

返回值：指向 struct tm 分段日期结构类型的指针。

#### 例程 14-7 将日历时间转换为 GMT。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    time_t t;
    struct tm *gmt;
    t=time(NULL);
    gmt=gmtime(&t);
    printf("GMT is:%s", asctime(gmt));
    return 0;
}
```



```
}
```

例程说明:

(1) 首先利用 `time` 函数获取系统时间, 它是一个日历时间。

(2) 再应用 `gmtime` 函数将该日历时间转换为格林尼治标准时间 (GMT)。该函数返回一个指向 `struct tm` 分段日期结构类型的指针。

(3) 最后应用 `asctime` 函数将分段日期转换成规定格式的字符串表示。

本例程的运行结果为:

```
GMT is:Sat Nov 10 06:25:13 2007
```

## 14.8 localtime 把日期和时间转换为结构

函数原型: `struct tm *localtime(const time_t *timer);`

头文件: `#include <time.h>`

是否是标准函数: 是

函数功能: 把 `timer` 所指的日历时间转换为以本地时间表示的分段时间。

返回值: 指向 `struct tm` 分段日期结构类型的指针。

**例程 14-8** 利用函数 `localtime` 将日历时间转换为分段时间。

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
int main(void)
{
    time_t timer;
    struct tm *tblock;
    timer = time(NULL);
    tblock = localtime(&timer);
    printf("Local time is: %s\n", asctime(tblock));
    printf("Today's date and time: %s\n", ctime(&timer)) ;
    getchar();
    return 0;
}
```

例程说明:

(1) 首先利用 `time` 函数获取系统时间, 它是一个日历时间。

(2) 利用函数 `localtime` 将获取的系统时间转换为分段时间 `tm`。

(3) 利用函数 `asctime` 将该分段时间转换为规定的字符串格式, 并显示。

(4) 利用函数 `ctime` 直接将日历时间转换为规定的字符串格式, 并显示。

本例程的运行结果为:

```
Local time is: Sat Nov 10 01:47:59 2007
Today's date and time: Sat Nov 10 01:47:59 2007
```

注意: 前面讲过, 函数调用 `asctime(localtime(&time))` 等价于函数调用 `ctime(&time)`, 其中 `&time` 为日历时间数据的指针。因此, 本例程中的两种输出方式结果是一样的。



头文件 `stdlib.h` 中定义了一些存储分配函数和一些杂项函数，本章将介绍这些函数。

## 15.1 `calloc` 分配主存储器函数

函数原型: `void *calloc(size_t nelem, size_t size);`

头文件: `#include <stdlib.h>`

是否是标准函数: 是

函数功能: 分配并刷新内存。函数 `calloc` 有两个参数, `nelem` 指出要分配内存空间的项数; `size` 表明每一项的字节数。

返回值: 分配成功返回第一个分配字节的指针, 否则返回 `NULL`。

### 例程 15-1 利用函数 `calloc` 动态分配内存空间。

```
#include <stdlib.h>
main()
{
    int i, j, *p=NULL;
    printf("Please enter the size for allocation\n");
    scanf("%d", &i);
    p=(int *)calloc(i, sizeof(int));
    if(p)
    {
        printf("Please enter %d datas\n", i);
        for(j=0; j<i; j++)
            scanf("%d", &p[j]);

    }
    else {
        printf("Allocation is fail\n");
        return 0;
    }
    printf("The datas are\n");
    for(j=0; j<i; j++)
        printf("%d ", p[j]);
}
```

例程说明:

- (1) 本例程首先通过终端输入要开辟内存空间的大小, 将其保留在变量 `i` 中。
- (2) 通过函数 `calloc` 动态分配内存空间 (由用户指定 `i`), 并将分配空间的首地址赋值给指针变量 `p`。
- (3) 如果分配成功 (`p` 不为 `NULL`), 向该内存空间写入数据。



(4) 打印出刚才写入的数据。

本例程的运行结果为：

```
Please enter the size for allocation
3
Please enter 3 datas
3 2 1
The datas are
3 2 1
```

注意：本例程中分配内存空间时采用动态分配函数 `calloc`，这样分配存储空间的方法同数组不同，可以由用户在程序中指定分配空间的大小，而不用像数组那样在程序编译前指定数组的大小。

## 15.2 malloc 动态分配内存函数

函数原型：void \*malloc(unsigned size);

头文件：#include<stdlib.h>

是否是标准函数：是

函数功能：动态分配一块内存空间，size 为指定的分配空间的大小（字节数）。

返回值：分配成功返回指向分配内存的指针，否则返回 NULL。

### 例程 15-2 利用函数 malloc 动态分配内存空间。

```
#include<stdlib.h>
main()
{
    char *str;
    if ((str = malloc(15)) == NULL)
    {
        printf("Not enough memory to allocate buffer\n");
        exit(1);
    }
    strcpy(str, "Hello World!");
    printf("String is %s\n", str);
    free(str);
    return 0;
}
```

例程说明：

(1) 首先利用函数 `malloc` 分配一个 15 字节大小的内存空间，并将其首地址赋值给指针型变量 `str`。

(2) 如果分配成功，复制字符串 "Hello World!" 到刚刚分配好的内存缓冲区中。

(3) 在屏幕上打印该字符串。

本例程的运行结果为：

```
String is Hello World!
```



## 15.3 realloc 重新分配主存函数

函数原型: `void *realloc(void *ptr, unsigned newsize);`

头文件: `#include<stdlib.h>`

是否是标准函数: 是

函数功能: 重新分配内存空间。`ptr` 为一个指针, 它指向重新设定大小的块; `newsize` 为重新分配内存的字节大小。

返回值: 分配成功返回修改块的指针, 否则返回 `NULL`。

### 例程 15-3 利用函数 realloc 重新分配内存空间。

```
#include<stdlib.h>
main()
{
    int *sqlist,i,len;
    len=10;
    sqlist=(int *)malloc(len*sizeof(int));
    for(i=0;i<20;i++)
    {
        if(i>=len){
            len=len*2;
            sqlist=realloc(sqlist,len*sizeof(int));
        }
        sqlist[i]=i;
    }
    for(i=0;i<20;i++)
        printf("%d ",sqlist[i]);
}
```

#### 例程说明:

- (1) 本例程首先分配一个只有 10 个整型数据大小的内存空间。
- (2) 然后通过程序向该数组输入 20 个整数。在这里要加一个判断, 即当输入的数据超过原来分配的内存空间的长度时, 调用 `realloc` 函数将内存重新分配, 大小为上一次长度的 2 倍。这样第  $i$  次调用 `realloc` 函数时分配的内存长度为  $10 \times 2^i$  个整型变量长度。
- (3) 最后打印这 20 个整数。

本例程的运行结果为:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

#### 注意:

- (1) `realloc` 函数的作用实际上分为两步, 一是在内存中重新开辟指定大小空间; 二是将原内存空间的数据复制到新开辟的空间中 (这是在新分配的内存比原内存大的情况下)。
- (2) 如果新分配的内存比原内存小, 则新分配的内存单元不被初始化。
- (3) `realloc` 函数多用于动态顺序表这种数据结构的建立。



## 15.4 free 释放内存函数

函数原型: void free(void \*ptr);

头文件: #include<stdlib.h>

是否是标准函数: 是

函数功能: 释放已分配的块。参数 ptr 为指向要释放的内存块的指针。

返回值: 无返回值。

### 例程 15-4 利用函数 free 释放内存空间。

```
#include<stdlib.h>
main()
{
    char *p;
    p=(char *)malloc(10*sizeof(char));
    strcpy(p,"Hello world\n");
    printf("%s",p);
    free(p);
    p=NULL;
    printf("%s",p);
}
```

例程说明:

- (1) 首先应用 malloc 函数在内存中分配一个 10 字节大小的内存空间。
- (2) 将字符串 "Hello world\n" 复制到该内存空间中, 并打印该字符串。
- (3) 释放掉 p 所指向的内存空间, 将 NULL 赋值给 p。
- (4) 显示 p 所指向的内容。

本例程的运行结果为:

```
Hello world
(null)
```

注意: 在使用完 malloc 或 calloc 函数分配的内存单元后, 应该用 free 函数释放掉分配的内存单元。特别是在开发一些大型系统时, 这样会节省资源。

## 15.5 abort 异常终止进程函数

函数原型: void abort(void);

头文件: #include<stdlib.h>

是否是标准函数: 是

函数功能: 异常终止一个进程, 并打印一条终止信息 Abnormal program termination 到 stderr。

返回值: 无返回值。

### 例程 15-5 利用 abort 函数终止一个程序。

```
#include <stdio.h>
```



```
#include <stdlib.h>
int main(void)
{
    printf("Calling abort()\n");
    abort();
    printf("Is the program be held?\n");
    return 0;
}
```

#### 例程说明:

本例程调用 `abort` 函数实现了一个程序的终止。实际上, 该程序执行到 `abort` 语句处就被异常终止了, 并没有执行后面两条语句。

本例程的运行结果为:

```
Calling abort()
Abnormal program termination
```

注意: 由于 `abort` 函数是异常终止一个进程, 因此系统会将一条终止信息 `Abnormal program termination` 输出到 `stderr`。

## 15.6 exit 正常终止进程函数

函数原型: `void exit(int status);`

头文件: `#include<stdlib.h>`

是否是标准函数: 是

函数功能: 正常终止一个进程。参数 `status` 用来保存调用进程的出口状态, 一般地, 0 表示正常退出, 非 0 表示发生错误。

返回值: 无返回值。

### 例程 15-6 应用 `exit` 函数正常终止一个程序。

```
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>
int main(void)
{
    float a,b;
    printf("Enter a\n");
    scanf("%f",&a);
    printf("Enter b\n");
    scanf("%f",&b);
    if(b)
        printf("a/b=%f",a/b);
    else
    {
        printf("Error:Divide by 0") ;
        exit(0);
    }
    getch();
    return 1;
}
```



**例程说明:**

- (1) 首先程序提示输入两个数  $a$  和  $b$ ，然后进行除法运算  $a/b$ 。
- (2) 如果  $b$  不为 0 表明运算合法，输出  $a/b$  的结果。
- (3) 如果  $b$  为 0，则运算不合法，于是在终端提示错误信息 "Error:Divide by 0"，并应用 `exit` 函数终止该程序。

本例程的运行结果为:

```
Enter a
3
Enter b
0
Error:Divide by 0
```

## 15.7 atexit 注册终止函数

**函数原型:** `int atexit(void * func);`

**头文件:** `#include <stdlib.h>`

**是否是标准函数:** 是

**函数功能:** 注册终止函数。其中参数 `func` 为指向函数的指针。在程序正常终止时，系统会调用注册了的 `func` 函数。

**返回值:** 注册成功返回 0，否则返回非 0。

### 例程 15-7 利用函数 `atexit` 注册出口函数。

```
#include <stdio.h>
#include <stdlib.h>
void fun1(void)
{
    printf("Exit function #1 called\n");
}

void fun2(void)
{
    printf("Exit function #2 called\n");
}

int main(void)
{
    atexit(fun1);
    atexit(fun2);
    return 0;
}
```

**例程说明:**

- (1) 首先程序依次注册（登记）了两个出口函数 `fun1` 和 `fun2`。
- (2) 当程序正常终止时，系统依次调用注册了的出口函数 `fun1` 和 `fun2`。

本例程的运行结果为:

```
Exit function #2 called
Exit function #1 called
```



注意:

(1) 最多可以用 atexit 函数注册 (登记) 32 个出口函数。

(2) 调用出口函数时, 按照先注册后调用, 后注册先调用的原则。因此本例程中先调用出口函数 fun2, 后调用出口函数 fun1。

## 15.8 getenv 获取环境变量

函数原型: `char *getenv(char *envvar);`

头文件: `#include <stdlib.h>`

是否是标准函数: 是

函数功能: 获得环境字符串的首地址。

返回值: 当 envva 所表示的环境变量存在时, 返回其首地址; 否则返回 NULL。

### 例程 15-8 显示环境变量。

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    char *s;
    s=getenv("COMSPEC");
    printf("Command processor: %s\n",s);
    return 0;
}
```

例程说明:

(1) 首先利用函数 getenv 获取名为 COMSPEC 的环境字符串的首地址。

(2) 根据得到的环境字符串的首地址, 显示该环境字符串。

本例程的运行结果为:

```
Command processor: C:\WINDOWS\SYSTEM32\COMMAND.COM
```

注意: C:\WINDOWS\SYSTEM32\COMMAND.COM 即为要得到的环境字符串。

## 15.9 bsearch 二分搜索函数

函数原型: `void *bsearch(const void *key, const void *base, size_t *nelem, size_t width, int(*fcmp)(const void *, const *));`

头文件: `#include <stdlib.h>`

是否是标准函数: 是

函数功能: 二分法查找。参数 key 指向要查找的关键字的指针, base 指向从小到大的次序存放元素的查找表, nelem 指定查找表元素的个数, width 指定查找表中每个元素的字节数, `int(*fcmp)(const void *, const *)` 为用户提供的比较函数。



返回值：如果没有找到匹配的值返回 0，否则返回匹配项的指针。

### 例程 15-9 用二分法在有序数列中查找元素。

```
#include <stdio.h>
#include <stdlib.h>
int CMP(int *a,int *b)
{
    if(*a<*b)
        return -1;
    else if(*a>*b)
        return 1;
    else
        return 0;
}
int main(void)
{
    int search[10]={1,3,6,7,10,11,13,19,28,56} ;
    int a=13,*p,i;
    /*对数组 search 进行二分搜索 13*/
    p=(int *)bsearch(&a, search,10, sizeof(int),CMP);
    printf("The elems of the array are\n");
    for(i=0;i<10;i++)
        printf("%d ",search[i]);
    /*显示元素 13 在原数组中的位置*/
    if(p)
        printf("\nThe elem 13 is located at %d of the array\n",p-search+1);
    else
        printf("\nSearch is fail!!\n");
    getchar();
}
```

#### 例程说明：

(1) 首先初始化一个查找数组 search，在这里，该数组一定是按照键值从小到大排列的。

(2) 利用函数 bsearch 进行二分法搜索。参数 &a 为要查找的关键字的指针，要查找的关键字 a 为 13；参数 search 为查找表首地址；10 为查找表元素个数；sizeof(int) 为查找表中每个元素的字节数，大小为 2 字节；CMP 为比较函数的指针。

(3) 将查找后的结果（匹配项的指针）赋值给指针变量 p。

(4) 显示元素 13 在原数组中的位置（由指针 p 和数组首地址 search 确定）。

本例程的运行结果为：

```
The elems of the array are
1 3 6 7 10 11 13 19 28 56
The elem 13 is located at 7 of the array
```

#### 注意：

##### (1) 关于用户提供的比较函数

bsearch 函数需要用户提供一个比较搜索函数，该函数由 bsearch 函数调用，并向该比较搜索函数传递两个指针参数 a 和 b。用户定义的比较函数必须在  $a < b$  时返回 -1，在  $a = b$  时返回 0，在  $a > b$  时返回 1。这里的大于、小于、等于，完全由用户来定义，与数学上的等价。



## (2) 二分法搜索简介

二分法搜索又叫做折半搜索或折半查找。它是一种经典的顺序文件查找算法，要求查找表按关键字有序排列（从小到大或从大到小，bsearch 函数要求从小到大排列）。其查找思想是：逐渐缩小查找范围，直至得到查找结果。查找过程为（以从小到大的序列为例）：将要查找的元素的關鍵字 k 与当前查找范围内位于居中的那个元素的關鍵字进行比较，若匹配，则查找成功，返回该元素的指针即可；若查找元素的關鍵字 k 小于当前查找范围内位于居中的那个元素的關鍵字，则到当前查找范围的前半部分重复上述查找过程，若查找元素的關鍵字 k 大于当前查找范围内位于居中的那个元素的關鍵字，则到当前查找范围的后半部分重复上述查找过程。

二分法搜索的查找效率较顺序搜索的查找效率要高许多，因此在进行顺序文件的搜索查找中是很实用的。有关二分法搜索算法的详细讲述请参看数据结构、算法分析等书目。

## 15.10 qsort 快速排序函数

函数原型：void qsort(void \*base, int nelem, int width, int(\*fcmp)(const void \*, const \*));

头文件：#include <stdlib.h>

是否是标准函数：是

函数功能：对记录进行从小到大快速排序。参数 base 指向存放待排序列的数组的首地址，nelem 为数组中元素的个数，width 为每个元素的字节数，int (\*fcmp)(const void \*, const \*) 为用户提供的比较函数。

返回值：无。

### 例程 15-10 利用 qsort 函数对无序序列进行快速排序（从小到大排序）。

```
#include <stdio.h>
#include <stdlib.h>
int CMP(int *a, int *b)
{
    if(*a < *b)
        return -1;
    else if(*a > *b)
        return 1;
    else
        return 0;
}
int main(void)
{
    int sort[10] = {3, 2, 6, 12, 1, 7, -5, 9, 30, 16};
    int i;
    printf("\nThe array that is before sort\n");
    for(i = 0; i < 10; i++)
        printf("%d ", sort[i]);
    qsort(sort, 10, sizeof(int), CMP);
    printf("\nThe array that is after sort\n");
    for(i = 0; i < 10; i++)
        printf("%d ", sort[i]);
    getchar();
}
```

**例程说明:**

(1) 首先初始化待排序列 sort。

(2) 显示最初数组 sort 中的元素排列情况。

(3) 对数列 sort 进行快速排序, 这里 sort 为数组 sort 的首地址; 10 为数组 sort 的元素个数; sizeof(int) 为待排序列中每个元素的字节数, 大小为 2 字节; CMP 为用户定义的比较函数的指针。

(4) 最后显示出排序后 (从小到大排序) 数组 sort 中的元素排列情况。

**本例程的运行结果为:**

```
The array that is before sort
3 2 6 12 1 7 -5 9 30 16
The array that is after sort
-5 1 2 3 6 7 9 12 16 30
```

**注意:**

(1) 关于用户提供的比较函数

qsort 函数需要用户提供一个比较搜索函数。该函数由 qsort 函数调用, 并向该比较搜索函数传递两个指针参数 a 和 b。用户定义的比较函数必须在  $a < b$  时返回 -1, 在  $a = b$  时返回 0, 在  $a > b$  时返回 1。这里的大于、小于、等于, 完全由用户来定义, 与数学上的等价。

(2) 快速排序简介

快速排序是一种经典的高效排序算法。它选取待排序列中某个元素为基准, 按照该元素值的大小将整个序列划分为左右两个子序列, 其中左子序列的值小于或等于基准元素的值; 右子序列的值大于或等于基准元素的值。然后分别对两个子序列重复上述排序过程, 直至所有元素都排在相应位置为止。

有关快速排序算法的详细介绍, 请参看数据结构、算法分析等书目。



## 第 3 部分

### 经典 C 编程实例与常见试题解析

在前面的两部分中已经对 C 语言的基础知识和标准 C 函数库做了详细地介绍，在讲解过程中也举了大量的例程，并介绍了一些常用的算法，旨在更加清楚地阐述知识点。本部分在前两分部的基础之上，通过对一些 C 程序实例的讲解，使读者巩固加深 C 语言的使用，同时加深理解结构化程序设计的方法。另外，本部分还加入一些经典的 C 程序实例分析，其中涵盖了一些经典的解题算法和数据结构内容，这样更利于读者利用 C 语言编程工具解决一些实际问题，同时巩固基础，启迪思维。本部分共包括 3 章，第 16 章：C 语言常用算法；第 17 章：经典 C 编程实例；第 18 章：常见 C 语言试题解析。

通过本部分的学习，读者在掌握了 C 语言基础知识和一些常用 C 函数的基础上，能够提高实际的编程能力和分析解决复杂问题的能力，从而提高应试能力，同时更加深入地理解结构化程序设计的思想和方法，掌握一些经典的解题算法和数据结构的使用。



C 语言是一种面向过程的程序设计语言，它多应用于结构化的程序设计。掌握 C 语言的编程方法是掌握其他高级语言编程方法的基础。本章将对 C 语言编程实践中的一些重要概念进行说明，其中包括结构化程序设计的思想和方法、算法的介绍、常用数据结构的介绍。

## 16.1 结构化程序设计

结构化程序设计的方法是一种面向过程的程序设计方法，与当前普遍使用的面向对象的程序设计思想不同，结构化的程序设计更加注重解决问题本身。

结构化程序设计方法的基本思想是：自顶向下，逐步求精，模块化设计。也就是说，在解决一个实际问题时，从一点入手，按照问题的流程一步一步地解决，整个程序（工作流程）按照子功能的不同划分为不同的功能模块，同时将现实中复杂的问题分解为多个简单的子问题，逐步细化地解决。这样不但可以降低解决问题的难度，而且符合软件工程的要求，能够提高程序开发的效率。

下面通过一个实例理解结构化程序设计的方法。

### **例程 16-1** 应用结构化程序设计方法设计一个图书馆管理系统。

要利用结构化程序设计方法来设计开发软件，首先要从软件的功能分析，这一点与面向对象的程序设计思想有着本质的不同。

一个较为完整的图书馆管理系统起码要包括以下几个功能：

- (1) 用户注册功能。
- (2) 书目的查询检索功能。
- (3) 借阅功能。
- (4) 还书功能。
- (5) 预订图书功能。
- (6) 过期罚款功能。
- (7) 查阅借书历史记录功能。

在软件的开发过程中，一般可以用系统的结构功能图来描述上述需求，如图 16-1 所示。



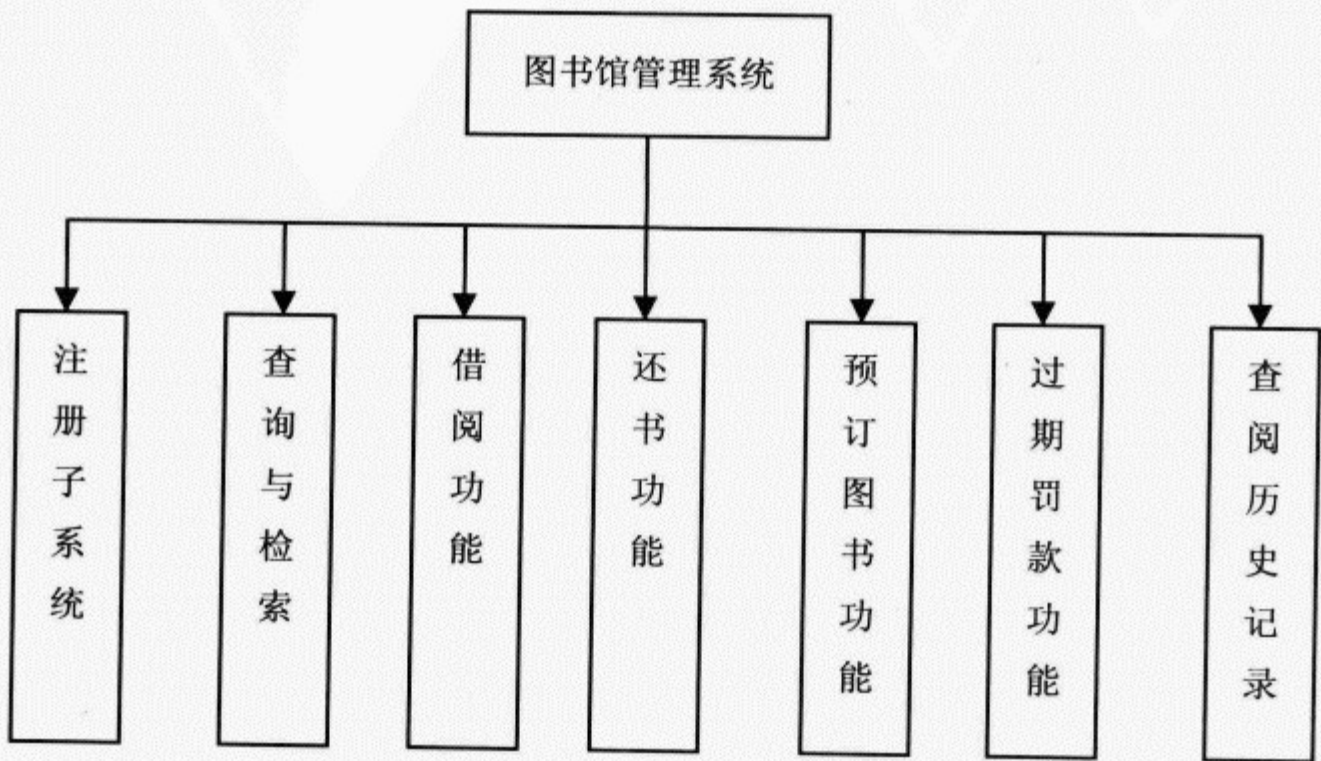


图 16-1 图书馆管理系统的结构功能图

这只是第一步的功能的细化，也正是通过这样一步简单的“分工”，将原来一个复杂的图书馆管理系统变得清晰了许多，接下来就是更进一步的考虑。例如在“注册子系统”中应当考虑给不同的用户提供不同的权限，如图 16-2 所示。

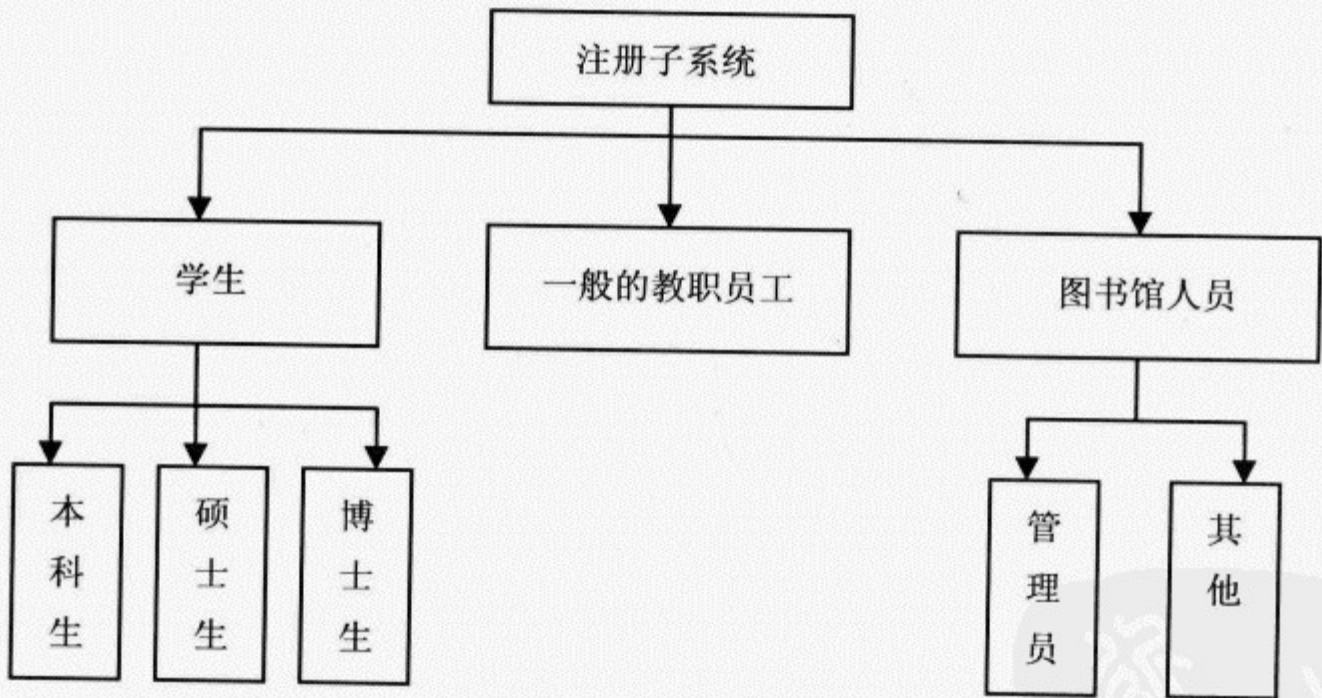


图 16-2 用户注册子系统

该系统要根据注册用户身份的不同提供不一样的权限。例如学生和老师的可借阅书籍的数目不一样；本科生、硕士生和博士生的可借阅书籍的数目也不一样；图书馆的工作人员与其他人员访问该系统的权限并不一样；而图书馆的工作人员中管理员与其他的图书馆的工作人员访问该系统的权限也不一样。

当然这只是对注册子系统的进一步细化，要真正开发一个功能较为完善的图书馆管

理系统要做的细化工作其实是很多的，这里不再赘述。这样一步步地分解下去就将原来一个庞大的图书馆管理系统分解成一个个子系统，从而简化了系统开发的复杂性。在程序设计过程中，再将这些细化好的子系统按照功能的不同映射成为不同功能的程序模块，由不同的程序员去编写，这样既使得程序本身结构性很强，便于理解和维护，又符合软件工程的要求，利于软件生产的效率。

在这个开发流程中，实际上就体现了一种结构化程序设计的思想。要解决的问题是实现一个图书馆管理系统，解决该问题的方法是，从该系统整体入手，自顶向下，逐步细化，同时将复杂的问题分解为简单的子问题，整个程序按照局部功能划分为不同的功能模块。

C语言是一个很好的结构化程序设计的开发工具（编程语言）。利用C语言开发软件的整个流程都是面向过程的。程序员在编写程序时更多考虑的是程序的流程以及程序功能的实现，而不是一个个实体对象的建立以及这些对象之间的消息传递（这是面向对象程序设计考虑的事）。另外，C语言中提供了函数调用机制，利用函数可以将一个庞大的程序从结构上划分为不同的功能模块，然后通过主调函数的调用实现特定的功能，因为在C语言中，模块是用函数来实现的。因此，在掌握了C语言基本知识的基础上，应当更多考虑如何使自己编写的程序更加符合结构化程序设计的要求。

## 16.2 程序的灵魂——算法

一个程序往往要包含两个方面的描述，一是对数据组织的描述；二是对程序操作流程的描述。对数据组织的描述主要是指数据的类型和数据的组织形式（例如数组），称作数据结构（data structure），在下一节将会讲到。对程序操作流程的描述就是程序的操作步骤，也是本节所要介绍的算法（algorithm）。正如 Nikiklaus Wirth 提出：

数据结构+算法=程序

一样，算法是一个程序中不可缺少的一部分。如果把一个可运行的程序比喻成一个具有生命的人，那么数据结构就是这个人的躯体，而算法则是这个人的灵魂或者说精神。

算法，广义地讲就是解决问题的方法和过程。例如洗衣服的过程就可描述为以下几步：

- （1）用盆接足量的清水。
- （2）将要洗的衣物浸入水中。
- （3）放入洗衣粉进行清洗。
- （4）用清水漂洗。
- （5）拧干衣物进行晾晒。

那么上述5步就可以叫做完成洗衣服这项工作的算法。

在计算机领域，算法更为严格的定义是若干条指令组成的有穷序列，它满足以下几条性质。

- （1）输入：有零个或多个外部提供的值作为算法的输入。
- （2）输出：至少产生一个量作为输出。
- （3）确定性：组成算法的每条指令确定无二义。



(4) 有限性：算法中每条指令执行的次数是有限的，每条指令的执行时间也是有限的。

以下讨论的算法都是指这种在计算机领域用于解决计算机要处理的问题的算法。  
下面从几个方面对算法进行讨论。

1. 算法的分类

计算机算法可分为两大类，一类叫做数值算法，它主要是解决一些工程上的数值计算问题，例如数值积分、数值求解微分方程等，《数值分析》和《计算方法》是专门讨论这种数值算法的。还有一类叫做非数值算法，它主要用于解决那些非数值的计算机问题。

2. 算法的表示

宽泛地讲，不管用哪种形式描述一个解题过程，只要它逻辑清晰，结果正确，即使是在脑子里构思的算法也是好的算法。但是实际问题往往比较复杂，需要用一种简单、清晰的“描述语言”来构建解题过程，这也就是算法的表示形式。

算法的表示形式很多，以下几种表示形式可供参考。

(1) 用自然语言描述

上面描述的洗衣服的过程就是一个用自然语言描述的算法。除非问题很简单，一般不用自然语言表示算法，因为自然语言存在二义性。

(2) 用流程图表示

这是一种最为普遍的算法表示方法，其主要优点是简单直观，便于理解。图 16-3 给出了流程图的图元表示方法。

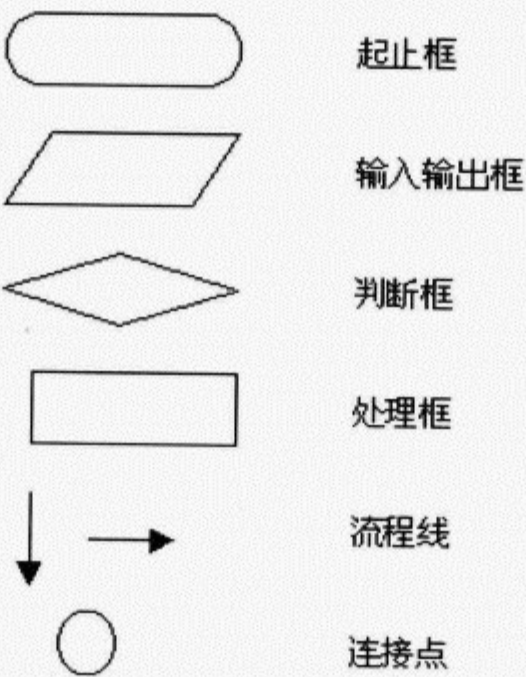


图 16-3 流程图的图元表示方法

下面通过一个例子来理解流程图表示的算法。



例程 16-2 用如图 16-4 所示流程图描述判断一个数  $i$  是否为素数。

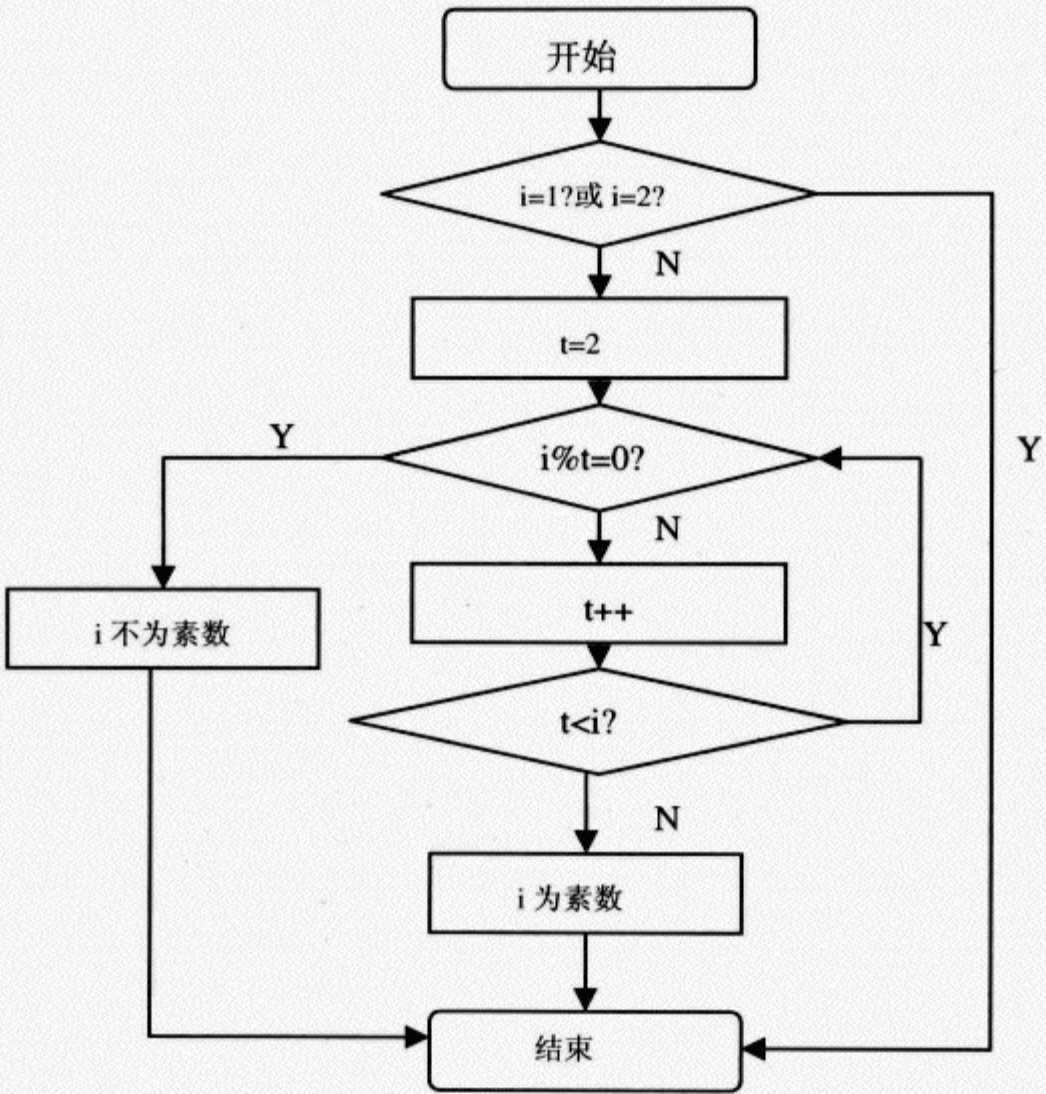


图 16-4 流程图的算法描述

(3) 用 N-S 流程图表示

1973 年美国学者提出了一种新型流程图：N-S 流程图。N-S 流程图表示方法如图 16-5 所示。

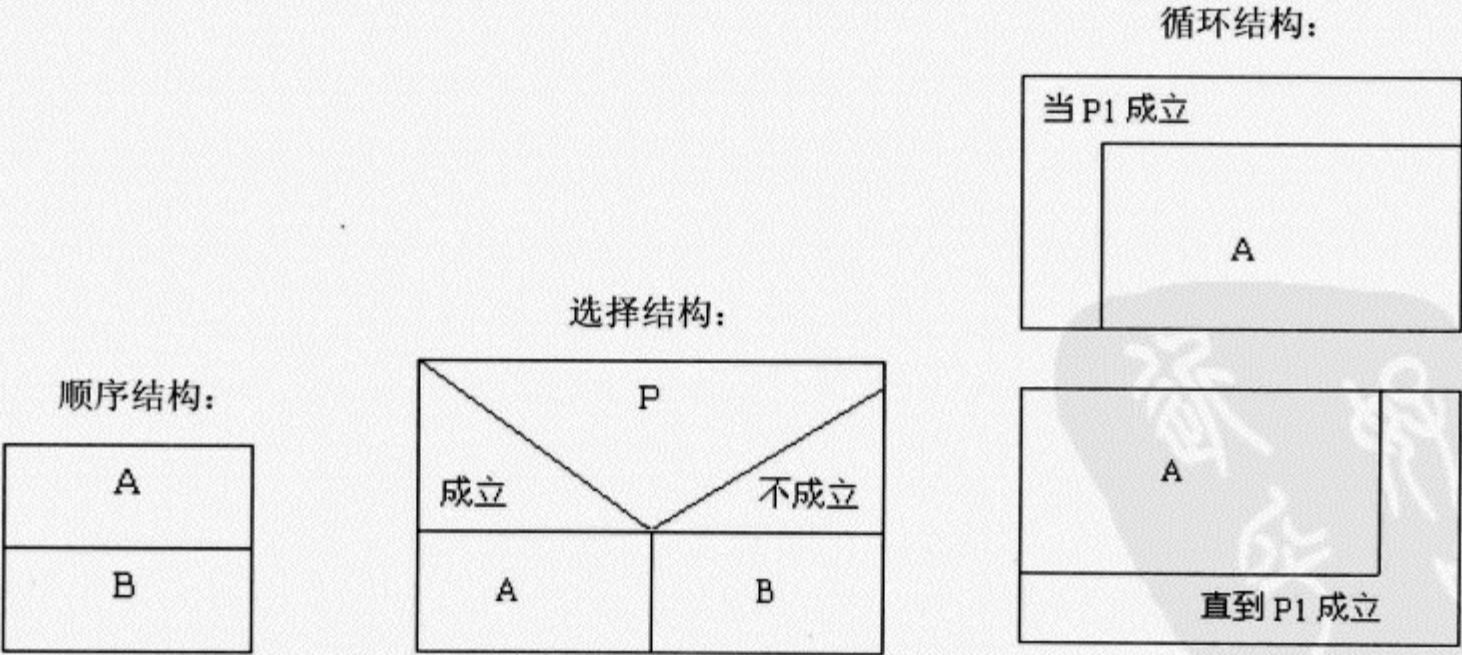


图 16-5 N-S 流程图的表示方法



#### (4) 用伪代码表示算法

伪码 (pseudocode) 是帮助程序员制定算法的智能化信息语言。伪代码使用介于自然语言和计算机语言之间的文字和符号来描述算法。伪代码程序并不能在计算机上运行, 它只作为程序员设计程序之前的一种辅助工具。因此伪代码并没有固定的语法和格式, 常根据程序员的习惯而定, 随意性很大。

#### 例程 16-3 用伪代码描述判断一个数 $i$ 是否为素数的算法。

```
If  $i=1$  or  $i=2$ 
Then  $i$  为素数, 程序结束
Else
 $T \leftarrow 2$ 
Repeat:
If  $i \bmod t = 0$ 
Then  $i$  不为素数, 程序结束
Else  $t \leftarrow t+1$ 
Until  $t \geq i$ 
 $i$  为素数, 程序结束
```

以上就是判断一个数  $i$  是否为素数的伪代码描述。不难看出, 伪代码描述较流程图算法描述更加接近程序代码形式, 因此也更容易转换为实际的程序。

### 3. 算法性能的测评

既然算法是解决实际问题的方法, 那么就必然存在着好坏优劣。一个算法的好坏是有指标能够加以测评的, 这个指标通常称为算法的复杂度。

算法的复杂度体现在运行该算法时所需要的系统资源开销上。如果计算机执行一个算法时所需要的系统资源开销很大, 就说这个算法复杂度很高; 相反, 如果计算机执行一个算法时所需要的系统资源开销很小, 就说这个算法复杂度很低。在设计算法时自然是希望算法的复杂度越低越好。

由于计算机最重要的资源是时间和空间资源, 因此, 算法的复杂度分为时间复杂度和空间复杂度。要想更进一步地探讨算法的复杂度问题, 可以参看《算法分析》等书目。

## 16.3 常用的数据结构

数据结构是指计算机内部数据的组织形式和存储方法。例如在前面章节中介绍的数组就是一种典型的线性数据结构。本章将更加具体地介绍一些常用的数据结构, 主要包括线性结构、树、图。

线性结构是最常用, 也是最简单的一种数据结构, 它是指由  $n$  个数据元素构成的有限序列。直观地讲, 它就是一张有限的一维数表。数组是一种最为简单的线性结构表示, 主要包括顺序表、链表、栈、队列等基本形式。其中顺序表和链表是从存储形式上来说的, 而栈和队列是从逻辑功能上来说的。顺序表和链表是线性数据结构的基础, 队列和栈基于顺序表和链表构成。

有时仅用线性结构存储管理数据是不够的, 一些数据之间存在着“一对多”的关系, 这就构成了树形结构, 简称树结构。例如人工智能领域常用的“博弈树”, 数据挖掘和商业智能中

使用的“决策树”、多媒体技术中的“哈夫曼树”等都是应用树形结构存储管理数据的实例。

还有一种常用的数据结构叫做图状结构，简称图结构。图结构中数据元素之间存在着“多对多”的关系，因此图结构较树结构、线性结构要复杂得多。在处理一些复杂的问题中，图结构往往能派上用场。例如人工智能领域中的“神经网络系统”、“贝叶斯网络”等都是应用图结构存储管理数据的实例。

本章重点介绍线性结构，对树结构和图结构只做简要的介绍。要想进一步学习数据结构的知识，可参看《数据结构》等书目。

## 16.4 顺序表

在计算机内部有不同方式存储一张线性表（线性结构的数表），最为简单方便的就是用一组连续地址的内存单元来存储整张线性表。这种存储结构称为顺序存储结构，这种存储结构下的线性表就叫做顺序表，如图 16-6 所示。

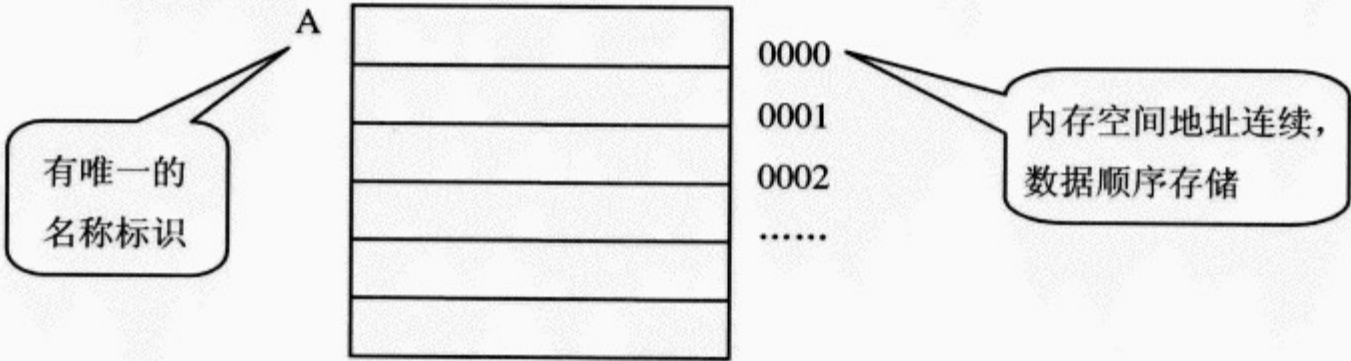


图 16-6 顺序表的示意图

从图 16-6 中不难看出，一张顺序表包括以下特征：

- (1) 有一个唯一的表名来标识该顺序表。
- (2) 内存单元连续存储，也就是说，一张顺序表要占据一块连续的内存空间。
- (3) 数据顺序存放，元素之间有先后关系。

因为数组满足上述特征，所以一个数组本身就是一张顺序表。下面介绍顺序表的定义以及对顺序表的几种操作。

### 16.4.1 顺序表的定义

定义一张顺序表也就是在内存中开辟一段连续的存储空间，并给它一个名字来标识。只有定义了一个顺序表，才能利用该顺序表存放数据元素，也才能对该顺序表进行各种操作。

顺序表有两种定义方法，一是静态地定义；二是动态生成。

静态地定义一张顺序表的方法与定义一个数组的方法类似。可以描述如下：

```
#define MaxSize 100
ElemType Sqlist[MaxSize];
int len;
```



为了更完整、更精确地描述一张顺序表，可以把顺序表定义成上面的样子。其中 `ElemType` 是指定的顺序表的类型，这里只是一个形式化的描述，要根据实际情况决定 `ElemType` 的内容。`ElemType` 可以是 `int`、`char` 等基本类型，也可以是其他构造类型。`len` 为顺序表的长度，定义这个变量目的是方便对顺序表的操作。

动态生成一张顺序表的方法可以描述如下：

```
#define MaxSize 100
typedef struct{
    ElemType *elem;
    int length;
    int listsize;
} SqList;
void initSqList(SqList *L){ /*初始化一个顺序表*/
    L->elem=(int *)malloc(MaxSize*sizeof(ElemType));
    if(!L->elem) exit(0);
    L->length=0;
    L->listsize= MaxSize;
}
```

动态生成一张顺序表时，首先定义一个类型 `SqList`，它是一个结构体，其成员包括指向顺序表的首地址，顺序表中表的长度（表中元素个数），顺序表的存储空间容量（占据内存大小，以 `sizeof(ElemType)` 为单位，由 `MaxSize` 规定）。

然后通过调用函数 `initSqList` 实现动态生成一张顺序表。函数 `initSqList` 的参数是一个 `SqList` 类型变量的引用，可以理解为在函数 `initSqList` 中直接对 `SqList` 类型变量 `L` 进行操作，这样省去了传递指针的麻烦。函数 `initSqList` 中通过 `malloc` 函数动态地分配一段长度为 `MaxSize*sizeof(ElemType)` 的内存空间。并将该段空间的首地址赋值给变量 `L` 的 `elem` 成员，也就是说 `L.elem` 指向顺序表的首单元。然后将 `L.len` 置为 0，表明此时刚刚生成一张空的顺序表，表内尚无内容；将 `L.listsize` 置为 `MaxSize`，表明该顺序表占据内存空间大小为 `MaxSize`（以 `sizeof(ElemType)` 为单位）。

#### 16.4.2 向顺序表中插入元素

下面介绍如何在长度为 `n` 的顺序表中第 `i` 个位置插入新元素 `item`。

在长度为 `n` 的顺序表中第 `i` 个位置插入新元素是指在顺序表第 `i-1` 个数据元素和第 `i` 个数据元素之间插入一个新元素 `item`。例如顺序表为：

$$A(a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n)$$

在第 `i` 个位置插入新元素 `item` 后，该顺序表变为：

$$A(a_1, a_2, \dots, a_{i-1}, \text{item}, a_i, \dots, a_n)$$

而此时顺序表 `A` 的长度由 `n` 变为 `n+1`。

下面给出向静态顺序表中第 `i` 个位置插入元素 `item` 和向动态生成的顺序表中第 `i` 个位置插入元素 `item` 的代码描述。

向静态顺序表中第 `i` 个位置插入元素 `item`。

```
void InsertElem(ElemType SqList[], int &n, int i, ElemType item){
    /*向顺序表 SqList 中第 i 个位置插入元素 item，该顺序表原长度为 n*/
```



```

int t;
if(n==MaxSize||i<1||i>n+1)
    exit(0);          /*非法插入*/
for(t=n-1;t>=i-1;t--)
    Sqlist[t+1]=Sqlist[t]; /*将 i-1 以后的元素顺序后移一个元素的位置*/
Sqlist[i-1]=item;      /*在第 i 个位置上插入元素 item*/
n=n+1;                /*表长加 1*/
}

```

首先判断插入元素的位置是否合法。一个长度为  $n$  的顺序表的可能插入元素的位置是  $1 \sim n+1$ ，因此  $i < 1$ 、 $i > n+1$  或者表满  $n = \text{MaxSize}$ （因为表的内存大小固定不变）的插入都是非法的。

然后将顺序表的  $i-1$  以后的元素顺序后移一个元素的位置，即将顺序表从第  $i$  个元素到第  $n$  个元素顺序后移一个元素的位置。

最后在表的第  $i$  个位置（下标为  $i-1$ ）上插入元素  $\text{item}$ ，并将表长加 1。

向动态生成的顺序表中第  $i$  个位置插入元素  $\text{item}$ 。

```

Void InsertElem(Sqlist *L,int I,ElemType item){
    /*向顺序表 L 中第 i 个位置上插入元素 item*/
    ElemType *base,*insertPtr,*p;
    if(i<1||i>L->length+1) exit(0); /*非法插入*/
    if(L->length>=L->listsize)
    {
        base=(ElemType*)realloc(L->elem,(L->listsize+10)*sizeof(ElemType));
        /*重新追加空间*/
        L->elem=base; /*新基址*/
        L->listsize=L->listsize+100; /*存储空间增大 100 单元*/
    }
    insertPtr=&(L->elem[i-1]); /* insertPtr 为插入位置*/
    for(p=&(L->elem[L->length-1]);p>=insertPtr;p--)
        *(p+1)=*p; /*将 i-1 以后的元素顺序后移一个元素的位置*/
    *insertPtr=item; /*在第 i 个位置上插入元素 item */
    L->length++; /*表长加 1*/
}

```

上述算法与“向静态顺序表中第  $i$  个位置插入元素  $\text{item}$ ”的算法类似，都是将  $i-1$  以后的元素顺序后移一个元素的位置，然后再向第  $i$  个位置上插入元素  $\text{item}$ 。不同之处在于向静态顺序表插入元素时，由于表的内存大小固定不变，所以只能在  $\text{MaxSize}$  规定的范围之内顺序插入元素。而向动态生成的顺序表中第  $i$  个位置插入元素  $\text{item}$  时，由于顺序表是建立在动态存储区的，可以随时扩充，因此当向表尾插入元素时，如果顺序表已满（ $L.\text{len} \geq L.\text{listsize}$ ），可以追加一段内存空间，再并入原顺序表中。

### 16.4.3 从顺序表中删除元素

下面介绍如何删除长度为  $n$  的顺序表中第  $i$  个位置的元素。

删除长度为  $n$  的顺序表中第  $i$  个位置的元素，就是将顺序表第  $i$  个位置上的元素去掉。例如顺序表为：

$$A(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$$

删除第  $i$  个位置的元素后，该顺序表变为：

$$A(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$$

而此时顺序表  $A$  的长度由  $n$  变为  $n-1$ 。



下面给出从静态顺序表中删除第  $i$  个位置元素和从动态生成的顺序表中删除第  $i$  个位置元素的代码描述。

从静态顺序表中删除第  $i$  个位置元素。

```
void DelElem(ElemType Sqlist[],int &n,int i){
    int j;
    if(i<1||i>n)
        exit(0)                /*非法删除*/
    for(j=i;j<n;j++)
        Sqlist[j-1]=Sqlist[j];    /*将第 i 位置以后的元素依次前移*/
    n--;                          /*表长减 1*/
}
```

首先判断要删除的元素是否合法。对于一个长度为  $n$  的顺序表，删除元素的合法位置是  $1 \sim n$ ，因此  $i < 1$  或者  $i > n$  都是不合法的。

然后将顺序表的第  $i$  位置以后的元素依次前移，这样就将第  $i$  个元素覆盖掉了，也就起到删除第  $i$  个位置元素的作用。

最后将表长减 1。

从动态生成的顺序表中删除第  $i$  个位置元素。

```
void DelElem(Sqlist *L,int i) {
    /*从顺序表 L 中删除第 i 个元素*/
    ElemType *delItem, *q;
    if(i<1||i>L->len) exit(0);          /*非法删除*/
    delItem=&(L->elem[i-1]);              /*delItem 指向第 i 个元素*/
    q=L->elem+L->length-1;                /*q 指向表尾*/
    for(++delItem; delItem<=q;++ delItem)*( delItem-1)=* delItem;
    /*将第 i 位置以后的元素依次前移*/
    L->length--;                          /*表长减 1*/
}
```

从动态生成的顺序表中删除第  $i$  个位置元素的方法与从静态顺序表中删除第  $i$  个位置元素的方法本质上是一样的，都是将第  $i$  个位置以后的元素依次前移，从而覆盖第  $i$  个元素。

顺序表是最简单的一种线性存储结构，它的优点是构造简单、操作方便，通过顺序表的首地址（表名）可直接对表进行随机存取，因此存取速度快，系统开销小。同时它也有着缺点：有可能浪费存储空间，在插入或删除一个元素时，需要对插入或删除位置后面的所有元素逐个进行移动，从而导致操作效率较低。因此，顺序表数据结构适用于表的长度不经常发生变化的场合，例如批处理。

下面通过程序实例来理解顺序表的应用。

#### 16.4.4 程序举例

例程 16-4 编写一个程序，创建一个顺序表。要求：顺序表初始长度为 10，向顺序表中输入 15 个整数，并打印出来；再删除顺序表中的第 5 个元素，打印出删除后的结果。

##### 例程 16-4 顺序表的应用。

```
#include "stdio.h"
#include "conio.h"
#define MaxSize 10
typedef int ElemType ; /*将 int 定义为 ElemType*/
```



```

typedef struct{
int *elem;
int length;
int listsize;
} Sqlist;

void initSqlist(Sqlist *L){ /*初始化一个顺序表*/
    L->elem=(int *)malloc(MaxSize*sizeof(ElemType));
    if(!L->elem) exit(0);
    L->length=0;
    L->listsize= MaxSize;
}

void InsertElem(Sqlist *L,int i,ElemType item){
    /*向顺序表L中第i个位置上插入元素item*/
    ElemType *base,* insertPtr,*p;
    if(i<1||i>L->length+1) exit(0);
    if(L->length>=L->listsize)
    {
        base=(ElemType*)realloc(L->elem,(L->listsize+10)*sizeof(ElemType));
        L->elem=base;
        L->listsize=L->listsize+100;
    }
    insertPtr=&(L->elem[i-1]);
    for(p=&(L->elem[L->length-1]);p>= insertPtr;p--)
        *(p+1)=*p;
    * insertPtr=item;
    L->length++;
}

void DelElem(Sqlist *L,int i) {
    /*从顺序表L中删除第i个元素*/
    ElemType *delItem,*q;
    if(i<1||i>L->length) exit(0);
    delItem=&(L->elem[i-1]);
    q=L->elem+L->length-1 ;
    for(++delItem; delItem<=q;++ delItem)* ( delItem-1)=* delItem;
    L->length--;
}

main()
{
    Sqlist l;
    int i;
    initSqlist(&l); /*初始化一个顺序表*/
    for(i=0;i<15;i++)
        InsertElem(&l,i+1,i+1); /*向顺序表中插入1……15*/
    printf("\nThe content of the list is\n");
    for(i=0;i<l.length;i++)
        printf("%d ",l.elem[i]);/*打印出顺序表中的内容*/
    DelElem(&l,5); /*删除第5个元素,即5*/
    printf("\nDelete the fifth element\n");
    for(i=0;i<l.length;i++) /*打印出删除后的结果*/
        printf("%d ",l.elem[i]);
    getch();
}

```

程序首先用函数 `initSqlist` 动态地创建了一个顺序表, 初始化长度为 `MaxSize 10`。然后通过函数 `InsertElem` 动态地向顺序表中插入数据。这里通过一个循环, 每次向顺序表的第 `i+1` 的位置插入整数 `i+1`。然后打印出该顺序表中的值。在插入顺序表的过程中, 由于顺序表初始化长度为 10, 而要插入 15 个元素, 因此要调用 `realloc` 函数为顺序表重新分配内存空间。再应用 `DelElem` 函数删除表中第 5 个元素, 也就是 5, 最后打印出删除元素后该顺序表中的值。



本程序的运行结果是：

```
The content of the list is
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Delete the fifth element
1 2 3 4 6 7 8 9 10 11 12 13 14 15
```

## 16.5 链表

在 7.1 节中已对链表的概念做了简要的概述。与顺序表相同，链表也是一种线性表，它的数据组织形式是一维的。不同的是，链表的存储结构是用一组地址任意地存储单元存储数据的，它不像顺序表那样占据一段连续的内存空间，而是将存储单元分散在内存的任意地址上。在链表结构中，每个数据元素记录都存放在链表的一个结点（node）中，结点之间由指针将其连接在一起，这样就形成了一条如同“链”的结构。链表的结点可以是一个结构体类型元素，也可以是其他的构造类型元素。在链表的每个结点中，都有一个专门用来存放指针（地址）的域，用这个指针域来存放后继结点的地址，这样就起到了连接后继结点的目的。一条链表通常有一个“表头”，它是一个指针变量，用来存放第一个结点地址。此外，链表的最后一个结点的指针域要置空（Null），因为它没有后继结点。链表的结构如图 16-7 所示。

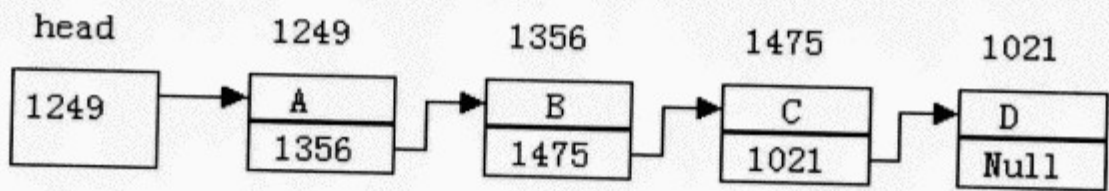


图16-7 链表结构示意图

从图 16-7 中可以看出链表存在以下特征。

- (1) 每一个结点包括两部分：数据域和指针域。其中数据域用来存放数据元素本身的信息，指针域用来存放后继结点的地址。
- (2) 链表逻辑上连续，物理上并不一定连续存储结点。
- (3) 只要获得链表的头结点，就可以通过指针遍历整条链表。

一个链表结点可用 C 语言描述如下：

```
typedef struct node{
    ElemType data;          /*数据域*/
    struct node *next       /*指针域*/
}Lnode,*LinkList;
```

这里采用自定义类型的方式将结构 struct node 定义为 Lnode 类型。也就是说该链表每个结点的类型为 Lnode。另外，\*LinkList 是指向 Lnode 类型数据的指针类型定义。也就是说在定义一个指向 Lnode 类型数据的指针类型变量时，语句

```
Lnode *L;
```

和

```
LinkList L;
```



是等价的。

下面介绍如何建立一个链表和操作链表。

### 16.5.1 创建链表

建立一个链表的过程可通过下面这段代码来描述。

```
LinkedList GreatLinkedList(int n){
    /*建立一个长度为n的链表*/
    LinkedList p,r,list=NULL;
    ElemType e;
    int i;
    for(i=1;i<=n;i++){
        Get(e);
        p=(LinkedList)malloc(sizeof(LNode));
        p->data=e;
        p->next=NULL;
        if(!list)
            list=p;
        else
            r->next=p;
        r=p;
    }
    return list;
}
```

上面这段代码描述了建立一条长度为  $n$  的链表的全过程。

首先用 `malloc` 函数在内存的动态存储区中开辟一块大小为 `sizeof(LNode)` 的空间,并将其地址赋值给 `LinkedList` 类型变量 `p` (`LinkedList` 为指向 `LNode` 变量的类型, `LNode` 为前面定义的链表结点类型)。然后将数据 `e` 存入该结点的数据域 `data`, 指针域存放 `NULL`。其中数据 `e` 由函数 `Get` 获得。

如果指针变量 `list` 为空,说明本次生成的结点为第一个结点,所以将 `p` 赋值给 `list`。`list` 是 `LinkedList` 类型变量,只用来指向第一个链表结点,因此它是该链表的头指针,最后要返回。

如果指针变量 `list` 不为空,则说明本次生成的结点不是第一个结点,因此将 `p` 赋值给 `r->next`。这里 `r` 是一个 `LinkedList` 类型变量,它始终指向原先链表的最后一个结点。

再将 `p` 赋值给 `r`,目的是使 `r` 再次指向最后的结点,以便生成链表的下一个结点,即保证 `r` 始终指向原先链表的最后一个结点。

最后将生成的链表的头指针 `list` 返回主调函数,通过 `list` 就可以访问到该链表的每一个结点,并对该链表进行操作。

### 16.5.2 向链表中插入结点

下面介绍如何在指针 `q` 指向的结点后面插入结点。

该过程的步骤如下:

- (1) 先创建一个新结点,并用指针 `p` 指向该结点。
- (2) 将 `q` 指向结点的 `next` 域的值赋值给 `p` 指向结点的 `next` 域。
- (3) 将 `p` 的值赋值给 `q` 的 `next` 域。

通过以上三步就可以实现在链表中在指针 `q` 指向的结点后面插入 `p` 所指向的结点。可以通过图 16-8 形象地展示这一过程。



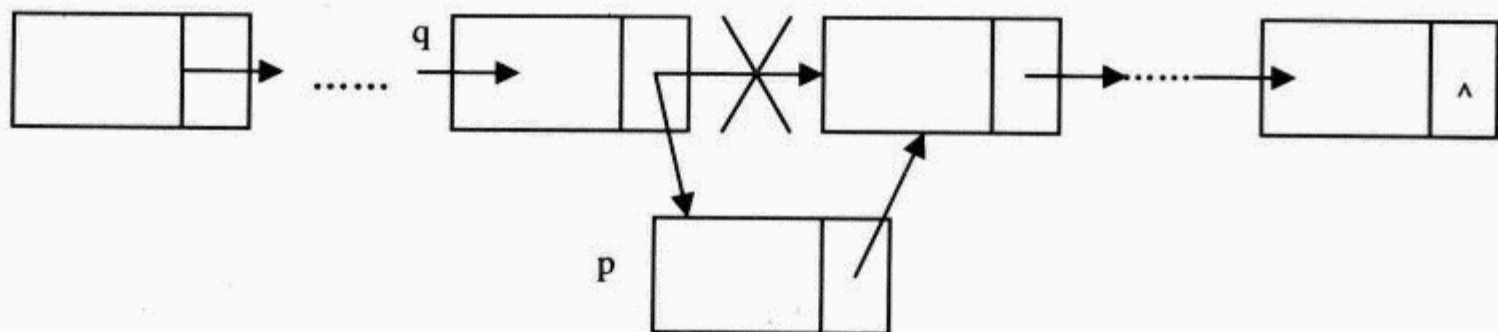


图 16-8 向链表插入结点过程

下面给出代码描述。

```
void insertList(LinkList *list, LinkList q, ElemType e) {
    /*向链表中在指针 q 指出的结点后面插入结点, 结点数据为 e*/
    LinkList p;
    p = (LinkList) malloc(sizeof(LNode));    /*生成一个新结点, 由 p 指向它*/
    p->data = e;                            /*向该结点的数据域赋值 e*/
    if (!*list) {
        *list = p;
        p->next = NULL;
    }
    /*当链表为空时*/
    else {
        p->next = q->next;
        /*将 q 指向的结点的 next 域的值赋值给 p 指向结点的 next 域*/
        q->next = p;
        /*将 p 的值赋值给 q 的 next 域*/
    }
}
```

### 16.5.3 从链表中删除结点

下面介绍如何从非空链表中删除 q 所指向的结点。

在讨论这个问题时, 必须考虑以下三种情形:

- (1) q 所指向的是链表的第一个结点。
- (2) q 所指向结点的前驱结点的指针已知。
- (3) q 所指向结点的前驱结点的指针未知。

下面给出不同情形的解决方法。

当 q 所指向的是链表的第一个结点时, 只需将 q 所指结点的指针域 next 的值赋值给头指针 list, 让 list 指向第二个结点, 再释放掉 q 所指结点即可, 如图 16-9 所示。

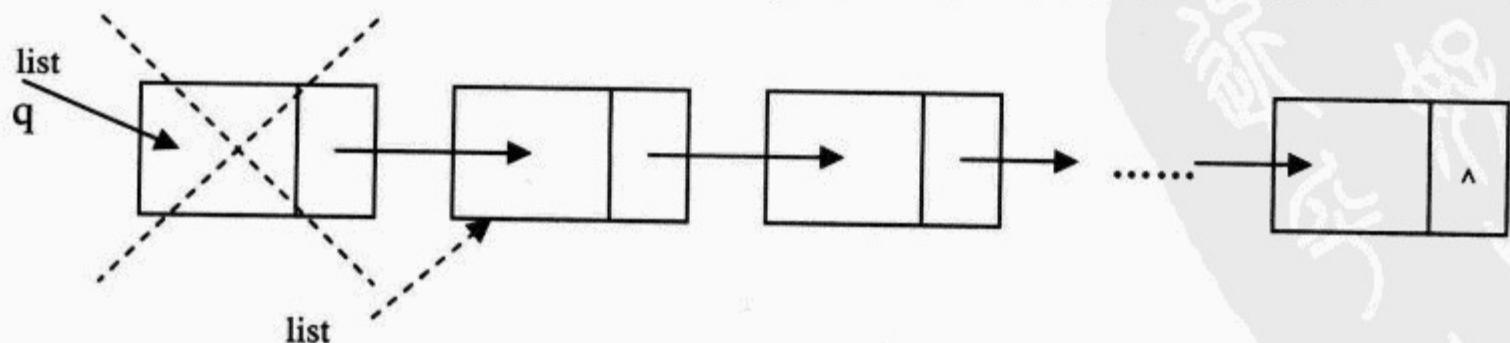


图 16-9 删除链表结点的第一种情形

当  $q$  所指向结点的前驱结点的指针已知时(假设为  $r$ ), 只需将  $q$  所指结点的指针域  $next$  的值赋值给  $r$  的指针域  $next$ , 再释放掉  $q$  所指结点即可, 如图 16-10 所示。

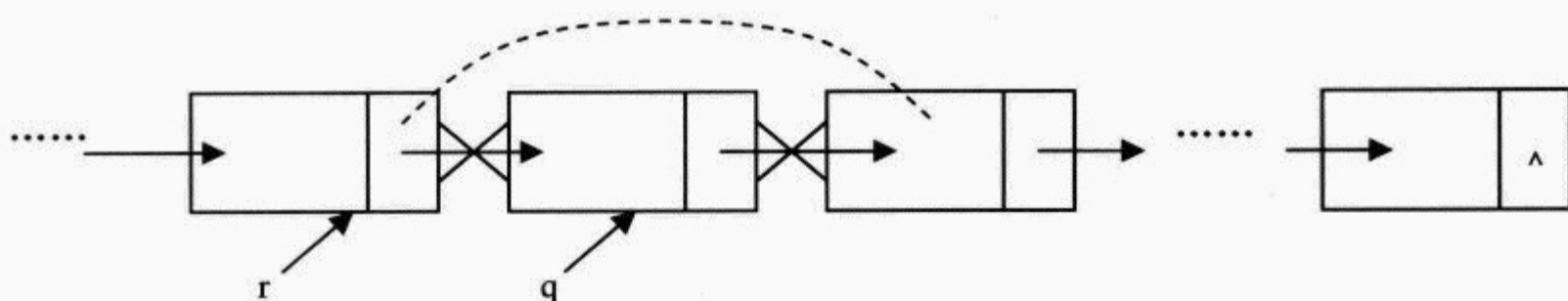


图 16-10 删除链表结点的第二种情形

以上两种情形可用下面这段代码来描述。

```
void delLink(LinkList *list, LinkList r, LinkList q) {
    if (q == *list)                /*删除链表结点的第一种情形*/
        *list = q->next;
    else
        r->next = q->next;        /*删除链表结点的第二种情形*/
    free(q);
}
```

当  $q$  所指向结点的前驱结点的指针未知时, 需要先通过链表头指针  $list$  遍历链表, 找到  $q$  的前驱结点的指针, 并将该指针赋值给指针变量  $r$ , 再按照第二种情形去做。

下面给出具体的代码描述。

```
void delLink(LinkList *list, LinkList q) {
    LinkList r;
    if (q == list) {
        *list = q->next;
        free(q);
    }
    else {
        for (r = *list; r->next != q; r = r->next); /*遍历链表, 找到 q 的前驱结点的指针*/
        if (r->next != NULL) {
            r->next = q->next;
            free(q);
        }
    }
}
```

使用链表方便灵活, 因为链表的结点是在内存的动态存储区中分配的, 可以根据程序的需要动态地为链表分配结点, 而且链表是逻辑上连续, 物理上不一定连续的线性存储结构, 因此它既具有线性结构逻辑简单、查找方便等优点, 又不过多地占用内存空间。

下面通过程序实例来理解链表的应用。

#### 16.5.4 程序举例

例程 16-5 编写一个程序, 要求: 从终端输入一组整数 (大于 10 个数), 以 0 作为结束标志, 将这一组整数存放在一个链表中 (结束标志 0 不包括在内), 打印出该链表中的值; 然后删除该链表中的第 5 个元素, 打印出删除后的结果。



**例程 16-5 键表的应用。**

```

#include "stdio.h"

typedef int ElemType;

typedef struct node{
    ElemType data; /*数据域*/
    struct node *next; /*指针域*/
}LNode,*LinkList;

LinkList GreatLinkList(int n){
    LinkList p,r,list=NULL;
    ElemType e;
    int i;
    for(i=1;i<=n;i++){
        scanf("%d",&e);
        p=(LinkList)malloc(sizeof(LNode));
        p->data=e;
        p->next=NULL;
        if(!list)
            list=p;
        else
            r->next=p;
        r=p;
    }
    return list;
}

void insertList(LinkList *list,LinkList q,ElemType e){
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode));
    p->data=e;
    if(!*list){
        *list=p;
        p->next=NULL;
    }
    else{
        p->next=q->next;
        q->next=p;
    }
}

void delLink(LinkList *list ,LinkList q){
    LinkList r;
    if(q==list){
        *list=q->next;
        free(q);
    }
    else{
        for(r=*list;r->next!=q;r=r->next);
        if(r->next!=NULL){
            r->next=q->next;
            free(q);
        }
    }
}

main()
{
    int e,i;
    LinkList l,q;

```



```

q=l=GreatLinkList(1); /*创建一个链表结点, q 和 l 指向该结点*/
scanf("%d",&e);
while(e) /*循环地输入数据, 同时插入新生成的结点*/
{
    insertList(&l,q,e) ;
    q=q->next;
    scanf("%d",&e);
}
q=l;
printf("The content of the linklist\n");
while(q) /*输出链表中的内容*/
{
    printf("%d ",q->data);
    q=q->next;
}
q=l;
printf("\nDelete the fifth element\n");
for(i=0;i<4;i++) /*将指针 q 指向链表第 5 个元素*/
{
    q=q->next;
}
delLink(&l,q); /*删除 q 所指的结点*/
q=l;
while(q) /*打印出删除后的结果*/
{
    printf("%d ",q->data);
    q=q->next;
}
getche();
}

```

本程序中, 首先应用函数 GreatLinkList 创建一个只含有 1 个结点的链表, 并向该结点中输入数据。然后通过函数 insertList 向链表中插入新的结点, 在插入结点的同时输入数据, 直到输入 0 为止, 然后打印出该链表中的数据。再通过循环使指针 q 指向该链表的第 5 个元素, 调用函数 delLink 删除 q 所指的结点, 最后打印出删除元素后链表中的值。

本程序的运行结果为:

```

1 2 3 4 5 6 7 8 9 10 11 0
The content of the linklist
1 2 3 4 5 6 7 8 9 10 11
Delete the fifth element
1 2 3 4 6 7 8 9 10 11

```

## 16.6 队列和栈

队列和栈是两种特殊的线性结构。它们基于链表和顺序表, 也就是说, 可以用链表或者顺序表来构造一个队列或一个栈。

队列就是一个先进先出 (first in first out, FIFO) 的线性表。它要求所有的数据从队列的一端进入, 从另一端离开。在队列中, 允许插入数据的一端叫做队尾 (rear), 允许数据离开的一端叫做队头 (front), 如图 16-11 所示。



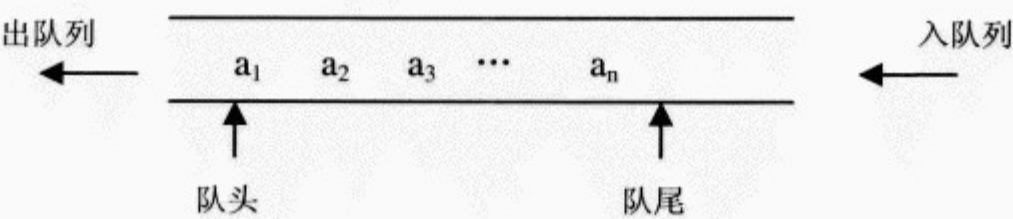


图 16-11  队列的示意图

对队列的操作主要包括队列的创建，数据入队列操作，数据出队列操作，清空队列，销毁队列等操作。在实际应用中，常用链表构造一个队列，称为链队列。只要保存两个链表指针就可以实现一个队列的功能，其中一个指向链表的头，作为队列的头指针；一个指向链表的尾，作为队尾指针。

栈是一种先进后出（last in first out，LIFO）的线性表。栈形如一个子弹膛，最先压进去的子弹一定最后弹出。栈是一种只限于在表尾端插入或删除的线性表，这个表尾称为栈的栈顶（top），表头称为该栈的栈底（bottom），如图 16-12 所示。

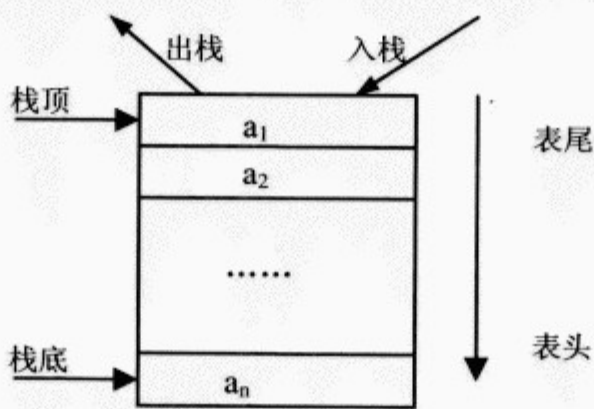


图16-12  栈结构的示意图

对栈的操作主要包括栈的构造，入栈操作，出栈操作，栈的判空操作（判断该栈是否为空），栈的判满操作（判断该栈是否已满）等。最简单的方法是用一个顺序表实现一个栈结构，称为顺序栈。设置变量 top 来指示栈顶元素在顺序栈中的位置，这样入栈出栈操作可通过变量 top 的自增自减来实现，也可以通过设置指向栈顶栈底的指针来实现对栈的操作，用法十分灵活。

队列和栈十分有用，利用栈可将许多递归程序转换为非递归程序，从而提高程序的执行效率，队列在一些大型的系统软件（例如操作系统）中使用十分广泛。

## 16.7  树结构

在程序设计中，树结构也是一种经常用到的数据结构。与线性结构不同，树结构采用的是非线性结构组织数据。在实际应用中，许多问题采用非线性结构来进行描述会更加简单、方便。

树结构是以分支关系定义的一种层次结构，因此应用树结构组织起来的数据应当具有层次关系，而具有这类特性的数据在计算机中应用十分广泛。例如操作系统中的文件管理，

网络系统中的域名管理，数据库系统中的索引，编译系统中的语法树等数据都是用树形结构组织的。

本节主要介绍树的概念、树结构的计算机存储形式及二叉树的概念。

### 1. 树的概念

树是由  $n$  ( $n \geq 0$ ) 个结点组成的有穷集合。在任意的一棵非空树中：

(1) 有且仅有一个称为根 (Root) 结点。

(2) 当  $n > 1$  时，其余的结点分为  $m$  ( $m > 0$ ) 个互不相交的有限集  $T_1, T_2, \dots, T_m$ ，其中每一个集合本身又是一棵树，称为根 (Root) 的子树 (SubTree)。

树的结构如图 16-13 所示。

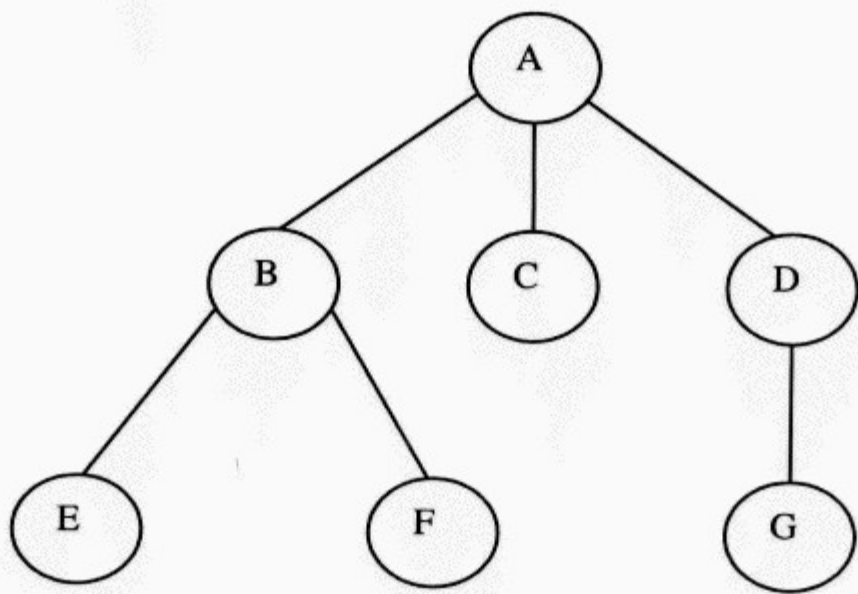


图16-13 树结构的示意图

图 16-13 中该树共有 7 个结点，其中 A 为根结点，即为该树的根；B, C, D 结点为根结点 A 的“孩子”，以 B, C, D 为根结点又构成三棵树，这三棵树为根结点 A 的子树，其中子树 C 只有一个根结点。

### 2. 树结构的计算机存储形式

树的最简单的一种存储形式为多重链表表示。在多重链表中，每个结点由一个数据域和若干个指针域组成，其中，每一个指针域的指针指向该结点的一个孩子结点。多重链表的结点类型可描述如下：

```
#define MaxChild 10
typedef struct node
{
    dataType data;
    struct node *child[MaxChild];
}
```

将树的每个结点都定义成如上所示的多重链表的结点。其中 data 为结点的数据域，存放结点的信息，child 为指向孩子结点的指针数组，通过这个指针数组将父结点与子结点联系起来。图 16-14 形象地描述了树的存储结构。



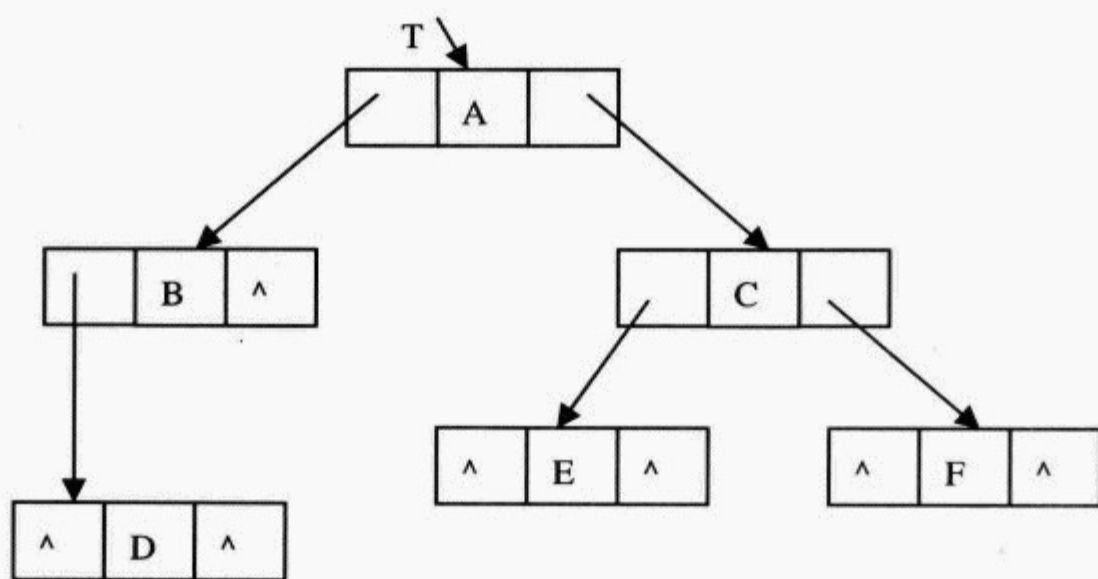


图16-14 树的存储结构

图 16-14 描述的是一棵二叉树（每个结点至多有两棵子树的树结构）的内部存储结构。可以看出，每个结点包含一个数据域和两个指针域。数据域用来存放结点中的数据信息，例如 A, B, C, ……。指针域用来存放指向孩子结点的指针，图中“^”表示空指针，表明无子结点。对于二叉树来说，一个结点至多可以有左右两个孩子。T 是一个指针，指向二叉树的根结点，只有得到了 T 这个指针才能访问整个树结构。

当然，树结构的存储形式不止这一种。在利用多重链表表示树结构时还分为定长结点表示和不定长结点表示两种。上面的例子是最为简单的定长结点表示，即每个结点的指针域个数固定（例中都是 2 个）。还有不定长的结点表示，即不同结点的指针域个数不同，这种表示方法最大的优点是节省内存资源，但操作起来相对麻烦。

### 3. 二叉树

二叉树是一种特殊形式的树结构。前面已经提到，二叉树的特点是每个结点最多有两棵子树，下面给出二叉树更为严格的定义。

二叉树（Binary Tree）是这样的树结构：它或者为空，或者由一个根结点加上两棵分别称为左子树和右子树的互不相交的二叉树组成。显然，这个定义是递归形式的。

在数据结构中，二叉树是树结构的研究重点。因为二叉树结构简单、易于数学抽象、计算机存储方便，另外任何一棵多叉树都可以通过逻辑上的转化变为二叉树形式存储，因此对二叉树的研究就显得更有意义。

这里只对树的概念、树的基本存储形式、二叉树的概念做一简单的介绍，要想深入理解树结构的相关知识可以参看《数据结构》等书目。

## 16.8 图结构

图是一种更为复杂的数据结构。在实际的程序设计中，数据元素之间存在着三种关系：一种是“先行后续”的关系，一个数据元素有一个直接前驱和一个直接后继，这种数据的组织结构叫做线性结构；一种是明显的层次关系，每一层上的数据元素可能和下一层中的

多个数据元素（孩子）相关，但只和上一层中的一个数据元素（双亲）相关，这种数据的组织结构叫做树结构；还有一种是数据元素之间存在“一对多”或者“多对一”的关系，也就是任意的两个数据元素之间都可以存在着关系，这种数据的组织结构叫做图结构。

1. 图的概念

图（graph）是由顶点的非空有限集合  $V$ （由  $N>0$  个顶点组成）与边的集合  $E$ （顶点之间的关系）所构成的。若图  $G$  中每一条边都没有方向，则称  $G$  为无向图；若图  $G$  中每一条边都有方向，则称  $G$  为有向图。

无向图如图 16-15 所示，有向图如图 16-16 所示。

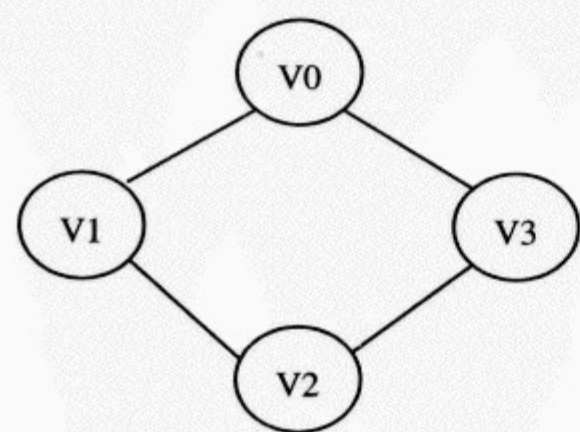


图16-15 无向图

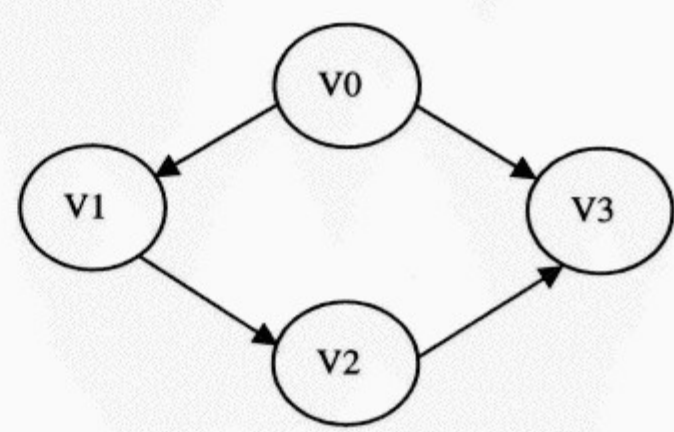


图16-16 有向图

2. 图结构的计算机存储形式

图最为常见的存储方法有两种：邻接矩阵存储方法和邻接表存储方法。

邻接矩阵存储方法也称数组存储方法，其核心思想是：利用两个数组来存储一个图。这两个数组一个是一维数组，用来存放图中的数据，一个是二维数组，用来表示图中顶点之间的相互关系，称为邻接矩阵。具体地，一个具有  $n$  个顶点的图  $G$ ，定义一个数组  $vertex[0,1,\cdots,n-1]$ ，将该图中顶点的数据信息分别存放在该数组中对应的数组元素上。例如：顶点  $v_0$  的数据信息存放在  $vertex[0]$  中，……，顶点  $v_i$  的数据信息存放在  $vertex[i]$  中。当然，数组  $vertex$  的类型要与图中顶点元素的类型一致。再定义一个二维数组  $A[0\cdots n-1][0\cdots n-1]$ ， $A$  称为邻接矩阵，它存放顶点之间的关系信息。 $A[i][j]$  定义为：

$$A[i][j]=\begin{cases} 1 & \text{当顶点 } i \text{ 与顶点 } j \text{ 之间有边} \\ 0 & \text{当顶点 } i \text{ 与顶点 } j \text{ 之间无边} \end{cases}$$

例如图 16-16 所示的有向图的邻接矩阵可表示为：

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

这样通过一个简单的邻接矩阵就可以把一个复杂的图关系表现出来，根据邻接矩阵



的信息 可以操纵数组 `vertex` 的元素，从而操纵整个图。

图 16-16 另一种存储形式是利用邻接表对图进行存储。邻接矩阵存储方法不适于存储稀疏图（边的数目很少的图），这时可以用邻接表存储这类图。

图 16-16 邻接表存储方法是一种顺序分配与链式分配相结合的存储方法。它由链表和顺序数组组成。链表用来存放边的信息，数组用来存放顶点的数据信息。具体地，对于图中的每一个顶点分别建立一个链表，如果一个图具有  $n$  个顶点，其邻接表就含有  $n$  个线性链表。每个链表前面设置一个头结点，称为顶点结点，结构如图 16-17 所示。

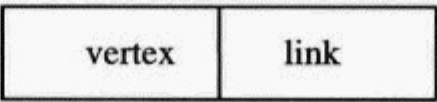


图 16-17 顶点结点的结构

顶点域 `vertex` 用来存放顶点的数据信息，指针域 `link` 指向依附于顶点 `vertex` 的第一条边。通常将一个图的  $n$  个顶点结点放到一个统一的数组中进行管理，并用该数组的下标表示顶点在图中的位置。

而在每一个链表中，链表的结点称之为边结点，它表示依附于对应的顶点结点的一条边。边结点的结构如图 16-18 所示。

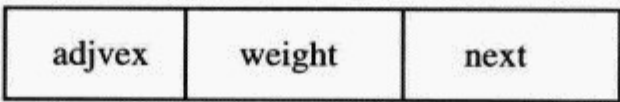


图16-18 边结点的结构

其中，`adjvex` 域存放该边的另一端顶点在顶点数组中的位置（数组下标）；`weight` 存放边的权重，对于无权重的图，此项省略；`next` 是指针域，它将第  $i$  个链表中所有边结点连接成一个链表，最后一个边结点的 `next` 域为 `NULL`。

图 16-16 所示的有向图的邻接表可表示为图 16-19 的形式。

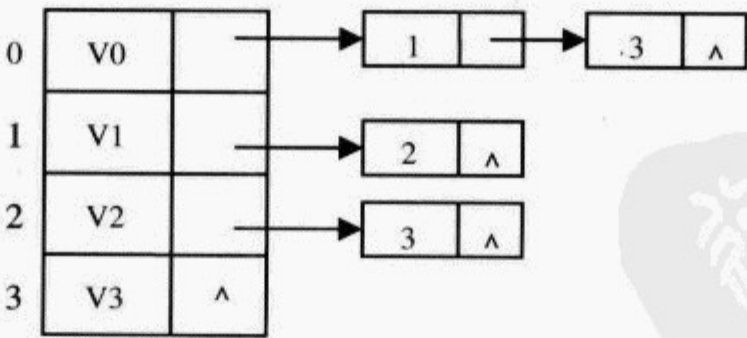


图16-19 图16-16的邻接表存储形式

这里只是对图的邻接矩阵存储法和邻接表存储法做一简要的介绍，有关其他形式（网的邻接矩阵表示、网络的邻接表、逆邻接表等）和它们的具体实现，有兴趣的读者可参看《数据结构》等书目。

为了使读者更好地掌握 C 语言编程技巧,巩固所学的 C 语言知识,深化结构化程序设计思想和面向过程的程序设计方法,本章列举了 12 个经典的 C 编程实例,内容涉及 C 语言的基础知识、常见问题的解决方法以及一些经典解题算法,难度由浅入深,逐步加大。相信本章的介绍能够对读者的编程能力给予有益的提升。

## 17.1 打印特殊图案

### 问题的提出

在应用 C 语言开发程序时,有时为了程序运行界面的美观,需要在屏幕上用字符构成一些特殊的图案来装饰。请设计一个 C 程序,实现在屏幕上输出一个类似于如图 17-1 所示的图案。

```
      *
     * * *
    * * * * *
```

图 17-1 \*字符构成的图案

### 问题的分析

这种关于特定格式输出的问题,最重要的一点是理解输出格式的规律。只要理解了输出格式的规律,通过简单的循环语句就可以得到希望的图案。

现在来分析一下上面这个图案输出格式的规律。仔细分析不难发现,上面的图案是由星号“\*”按照一定的规律逐行打印形成的一个三角形。共打印了三行“\*”字符,第一行一个“\*”,第二行三个“\*”,第三行五个“\*”。其中第一行的“\*”空出两个“\*”的位置,在第三个位置上打印出来,而且只打印一个;第二行的“\*”空出一个“\*”的位置,从第二个位置开始打印,共打印三个“\*”,第三行的“\*”就从起始位置开始打印,共打印五个“\*”。由此可以抽象出这样的规律:设共打印  $n$  行这样的“\*”构成一个等腰三角形,第  $i$  行的“\*”从第  $n-i+1$  的位置开始打印,共打印  $2i-1$  个“\*”,其中  $i=1,2,3\dots$ 。用这个规律来验证上面的图案显然是符合的。

有了上述规律,不但可以打印出像上面的由三行“\*”字符构成的等腰三角形,还可以打印出更多行的“\*”字符构成的等腰三角形,其行数可由程序来控制。



下面给出解决该问题的伪代码算法描述。

输入要打印等腰三角形图案的“\*”字符的行数  $n$ 。

```
I←1
repeat:
打印 (n-i) 个 space
打印 (2i-1) 个 "*"
打印 1 个 "\n"
I←i+1
until i>n
```

### 程序清单

```
/****** Procedure PrintTrangle******/
#include <string.h>
#include <stdio.h>
void PrintTrangle(int n);
int main()
{
    int n;
    printf("How many rows of * for trangle\n");
    scanf("%d",&n);
    PrintTrangle(n);
    getchar();
    return 0;
}
void PrintTrangle(int n)
{
    int i,j;
    for(i=1;i<=n;i++)
    {
        for(j=0;j<n-i;j++)
            printf(" ");    /*打印(n-i) 个 space*/
        for(j=0;j<2*i-1;j++)
            printf("*");    /*打印(2i-1) 个 "*"*/
        printf("\n");
    }
}
```

### 运行效果

$n=3$  时的输出效果。

```
How many rows of * for trangle
3
  *
 ***
*****
```

$n=8$  时的输出效果。

```
How many rows of * for trangle
8
      *
     ***
    *****
   *********
  ***********
 *****
*****
*****
*****
```

## 17.2 打印乘法口诀表

### 问题的提出

请在屏幕上输出一张乘法口诀表，形如：

1\*1=1

1\*2=2 2\*2=4

1\*3=3 2\*3=6 3\*3=9

1\*4=4 2\*4=8 3\*4=12 4\*4=16

1\*5=5 2\*5=10 3\*5=15 4\*5=20 5\*5=25

1\*6=6 2\*6=12 3\*6=18 4\*6=24 5\*6=30 6\*6=36

1\*7=7 2\*7=14 3\*7=21 4\*7=28 5\*7=35 6\*7=42 7\*7=49

1\*8=8 2\*8=16 3\*8=24 4\*8=32 5\*8=40 6\*8=48 7\*8=56 8\*8=64

1\*9=9 2\*9=18 3\*9=27 4\*9=36 5\*9=45 6\*9=54 7\*9=63 8\*9=72 9\*9=81

### 问题的分析

与上一节介绍的方法一样，现在来分析一下乘法口诀表的输出规律。一张乘法口诀表可以看作由81组算式构成的三角形数表。其中每个算式都是一个乘法式，并给出计算结果（例如  $3*8=24$ ）。仔细观察不难发现，每个算式的被乘数都等于该算式所处口诀表中的列数，每个算式的乘数都等于该算式所处口诀表中的行数。即如果一个算式为  $i*j=n$ ，那么该算式一定处在口诀表中的第  $j$  行，第  $i$  列， $n$  为  $i*j$  的计算结果。另外，口诀表的第  $j$  行中只输出  $j$  个乘法算式，共输出9行。

根据这个规律，就可以设计出输出一张乘法口诀表的算法。如果是按行输出该乘法口诀表，就可以应用一个二重循环语句。其中外层循环用变量  $j$  控制行数；内层循环用变量  $i$  控制列数，那么每一个输出的算式自然就是  $i*j=n$ 。

下面给出解决该问题的伪代码算法描述。

```
J ← 1, i ← 1
repeat:
    repeat:
        输出 i*j 及其结果
        i ← i+1
    until i > j
    打印 1 个 "\n"
    j ← j+1
until j > 9
```

；实现输出乘法口诀表的第  $j$  行

### 程序清单

```
/****** Procedure PrintMulTab *****/
#include <string.h>
#include <stdio.h>

void PrintMulTab();

int main()
{
```



```
PrintMulTab() ;
getchar();
}

void PrintMulTab()
{
    int i,j;
    for (j=1;j<=9;j++)
    {
        for(i=1;i<=j;i++)
            printf("%d*%d=%d ",i,j,j*i);
        printf("\n");
    }
}
```

运行效果

```
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6 3*3=9
1*4=4 2*4=8 3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
```

## 17.3 计算 100 以内的素数

### 问题的提出

在屏幕上输出 100 以内的所有素数。

### 问题的分析

素数是数学中一类非常重要的数字。素数是这样的自然数：除 1 以外，它只能被 1 和它本身整除。例如 2、3、5、7 等。

要输出 100 以内的所有素数，最简便的方法就是“穷举法”。穷举法就是举出问题答案的可能域中的所有可能值，然后对每一个值都进行判断，找出符合要求的答案。对本题而言，问题答案的可能域就是 1~100。因此，就可以穷举出 1~100 闭区间中的每一个数字（自然数），然后对每一个数字进行判断，看它是不是素数，如果是素数，则将该数输出；如果不是素数，则不输出该数。

这一过程的伪代码算法描述为：

```
i ← 1
repeat:
if i 是素数 then 输出 i endif
i ← i+1
until i > 100
```

下面要解决的就是如何判断  $i$  是否为素数。根据素数的定义可知，要判断  $i$  是否为素数，只需判断  $i$  除了 1 和  $i$  之外是否还能被其他数整除，如果  $i$  还能被其他数整除，则它一定不是素数；否则是素数。注意 1 不是素数。



这一过程的伪代码算法描述为:

```
n←2
while n<i
do:
    if i mod n=0
    then i 可被 n 整除, i 一定不是素数, 置标记, 跳出该循环。
    endif
    n←n+1
if 未置标记 then i 是素数
else i 不是素数 endif
```

最后可将上述两个算法映射成为两个函数, 通过函数的调用实现该功能。

### 程序清单

```
/****** Procedure Primes******/
#include <string.h>
#include <stdio.h>

void allPrimes(int l,int h);      /*输出从 l 到 h 以内全部的素数*/
int isPrime(int i);              /*判断 i 是否是素数*/

void main()
{
    printf("The Primes from 1 to 100 are:\n");
    allPrimes(1,100);
    getchar();
}

void allPrimes(int l,int h)
{
    int i;
    for(i=l;i<=h;i++)
        if(isPrime(i)) printf("%d ",i);
}

int isPrime(int i)
{
    int n,flag=1;
    if(1==i)flag=0;
    for(n=2;n<i;n++)
        if(i%n==0){flag=0;break;}
    if(flag==1)
        return 1;
    else
        return 0;
}
```

### 运行效果

```
The Primes from 1 to 100 are:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

## 17.4 判断回文数字

### 问题的提出

有这样一类数字, 它们顺着看和倒着看是相同的数, 例如 121, 656, 2332 等, 这样的



数字叫做回文数字。编写一个程序，判断从键盘接收的数字是否为回文数字。

### 问题的分析

要想判断一个数是否是回文数字，必须从回文数字的特点入手。因为回文数字顺着看和倒着看是相同的数，所以可以通过这个特点来判断一个数字是否是回文数字。

可以通过将一个十进制数“倒置”的办法来判断它是否是回文数字，倒置就是计算该十进制数倒过来后的结果。例如一个数是 123，它的倒置结果为 321，因为 123 不等于 321，所以 123 不是回文数字。同理，一个数是 121，它的倒置结果也为 121，所以 121 是回文数字。

这一过程的伪代码算法描述为：

```
输入一个十进制数 i
计算 i 的倒置数 j
if i=j then i 是回文数字
else i 不是回文数字
endif
```

本题的关键是如何计算一个十进制数  $i$  的“倒置数”  $j$ 。

假设一个十进制数为  $10^2a_1+10a_2+a_3$ ，显然它的倒置数应该是  $10^2a_3+10a_2+a_1$ 。因此关键是得到每一位上的数  $a_i$ ，过程可描述为：

(1) 将十进制数  $10^2a_1+10a_2+a_3$  模 10，得到个位数  $a_3$ ，并将十进制数  $10^2a_1+10a_2+a_3$  整除 10，得到新的十进制数字  $10a_1+a_2$ 。

(2) 将十进制数  $10a_1+a_2$  模 10，得到个位数  $a_2$ ，并将十进制数  $10a_1+a_2$  整除 10，得到新的十进制数  $a_1$ 。同时将  $a_3$  乘以 10 加上  $a_2$ ，得到  $10a_3+a_2$ 。

(3) 将十进制数  $a_1$  模 10，得到个位数  $a_1$ ，同时将  $10a_3+a_2$  乘以 10 加上  $a_1$ ，从而得到  $10^2a_3+10a_2+a_1$ ，即为倒置数。

因此，可以抽象出下面的算法来计算  $i$  的倒置数。

```
m ← i
j ← 0
repeat:
    j ← j*10 + (m MOD 10)
    m ← m 整除 10
until m ≤ 0
```

最后可将上述两个算法映射成为两个函数，通过函数的调用实现该功能。

### 程序清单

```
/****** Procedure isCircle *****/
#include <string.h>
#include <stdio.h>

int isCircle(int n); /*判断 n 是否是回文数字*/
int reverse(int i); /*计算 i 的倒置数*/

void main()
{
    int n;
    printf("Type a integer for judging is Circle:\n");
    scanf("%d",&n);
    if(isCircle(n))
```



```
        printf("%d is Circle\n",n);
    else
        printf("%d is not Circle\n",n);
    getchar();
}

int isCircle(int n)
{
    int m;
    m= reverse(n);
    if(m==n)
        return 1;
    else
        return 0;
}

int reverse(int i)
{
    int m,j=0;
    m=i;
    while(m){
        j=j*10+m%10;
        m=m/10;
    }
    return j;
}
```

### 运行效果

判断 123 不是回文数字:

```
Type a integer for judging is Circle:
123
123 is not Circle
```

判断 12321 是回文数字:

```
Type a integer for judging is Circle:
12321
12321 is Circle
```

## 17.5 计算最大公约数

### 问题的提出

计算两个数的最大公约数。

### 问题的分析

两个数最大公约数是指两个数  $a$ ,  $b$  的公共因数中最大的那一个。例如 4 和 8 的公共因数分别为 1、2、4，其中 4 为 4 和 8 的最大公约数。

因此要计算出两个数的最大公约数，首先要找到两个数的全部公共因数，然后再从里面挑出最大的那个公共因数即为欲求的最大公约数。

如果一个数  $i$  为  $a$  和  $b$  的公共因数，那么一定满足  $a\%i$  等于 0，并且  $b\%i$  等于 0。所以，



设计算法时只需穷举出小于等于  $\min(a,b)$  的全部正整数, 然后逐个判断它是否为  $a$  和  $b$  的公共因数。如果是  $a$  和  $b$  的公共因数就记录下来, 最后挑出最大的那个公共因数即为  $a$  和  $b$  的最大公约数。

这一过程的伪代码算法描述为:

```
i ← 1
max ← -1      ; 为最大公约数设置初值-1
repeat:
if a mod i 等于 0 并且 b mod i 等于 0 then    ; i 是 a,b 的公共因数
    if i > max then
        max ← i      ; 如果 i 大于 max, 则将 i 设为临时的最大公约数
    endif
i ← i + 1
endif
until i >= min(a,b)
max 为所求的 a 和 b 的最大公约数
```

### 程序清单

```
/****** Procedure GYS******/
#include <string.h>
#include <stdio.h>

int MaxElem(int a,int b);
void main()
{
    int a,b ,res;
    printf("Please input two integer\n");
    scanf("%d %d",&a,&b);
    res=MaxElem(a,b);
    printf("The MaxElem of %d and %d is %d\n",a,b,res);
    getchar();
}

int MaxElem(int a,int b)
{
    int i,max,min;
    if(a>b)min=b;
    else min=a;
    for(i=1;i<=min;i++)
        if(a%i==0&&b%i==0)
            max=i;
    return max;
}
```

### 运行效果

```
Please input two integer
16 24
The MaxElem of 16 and 24 is 8
```

## 17.6 寻找阿姆斯特朗数

### 问题的提出

如果一个正整数等于其各个数字的立方和, 则称这个数为阿姆斯特朗数, 例如:

$407=4^3+0^3+7^3$ ，因此 407 就是一个阿姆斯特朗数。编写一个程序，找出 1000 以内的所有阿姆斯特朗数。

### 问题的分析

同前面求解 100 以内的素数思路一样，应用穷举法穷举出 1~1000 闭区间中的每一个数字（正整数），然后对每一个正整数进行判断，看它是不是阿姆斯特朗数，如果是阿姆斯特朗数，则将该数输出；如果不是阿姆斯特朗数，则不输出该数。

这一过程的伪代码算法描述为：

```
i ← 1
repeat:
if i 是阿姆斯特朗数 then 输出 i endif
i ← i+1
until i > 1000
```

于是，问题归结到如何判断一个数是否是阿姆斯特朗数上。根据定义易知，要判断一个数是否是阿姆斯特朗数，只需判断该数是否等于其各个数字的立方和。又因为如果一个数为  $a$ ，则  $a\%10$  即为该数的个位数字， $a/10\%10$  即为该数的十位数字， $a/100\%10$  即为该数的百位数字，……，依此类推，因此，判断一个数  $a$  是否是阿姆斯特朗数的伪代码算法可描述为：

```
sum ← 0
tmp ← a
repeat:
    sum ← sum + (tmp mod 10)³
    tmp ← tmp / 10
until tmp ≤ 0
if sum 等于 a then
    a 是阿姆斯特朗数
else a 不是阿姆斯特朗数
endif
```

最后可将上述两个算法映射成为两个函数，通过函数的调用实现该功能。

### 程序清单

```
/****** Procedure Armstrong******/
#include <string.h>
#include <stdio.h>

int IsArmstrong(int a);
void Armstrong();
void main()
{
    printf("The Armstrong numbers below 1000 are\n");
    Armstrong();
    getchar();
}

void Armstrong()
{
    int i;
    for(i=0; i<=1000; i++)
        if(IsArmstrong(i))
            printf("%d ", i);
}
```



```

int IsArmstrong(int a)
{
    int sum=0,tmp;
    tmp=a;
    while(tmp>0)
    {
        sum=sum+(tmp%10)*(tmp%10)*(tmp%10);
        tmp=tmp/10;
    }
    if(sum==a)
        return 1;          /*a 是阿姆斯特朗数*/
    else
        return 0;          /* a 不是阿姆斯特朗数*/
}

```

运行效果

```

The Armstrong numbers below 1000 are
0 1 153 370 371 407

```

## 17.7 歌德巴赫猜想的近似证明

### 问题的提出

众所周知，歌德巴赫猜想的证明是一个世界性的数学难题，至今未能完全解决。我国著名数学家陈景润为歌德巴赫猜想的证明做过杰出的贡献。

歌德巴赫猜想是说任何一个大于 2 的偶数都能表示成为两个素数之和。应用计算机工具可以很快地在一定范围内验证歌德巴赫猜想的正确性。请编写一个 C 程序，验证指定范围内歌德巴赫猜想的正确性，也就是近似证明歌德巴赫猜想（因为不可能用计算机穷举出所有正偶数）。

### 问题的分析

问题归结为在指定范围内（例如 1~2000 内）验证其中每一个偶数是否满足歌德巴赫猜想的论断，即是否能表示为两个素数之和。如果发现一个偶数不能表示为两个素数之和，则意味着举出了反例，从而可以否定歌德巴赫猜想。

可以应用枚举的方法枚举出指定范围内的每一个偶数，然后判断它是否满足歌德巴赫猜想的论断，一旦发现有不满足歌德巴赫猜想的数据，则可以跳出循环，并做出否定的结论；否则如果集合内的数据都满足歌德巴赫猜想的论断，则可以说明在该指定范围内，歌德巴赫猜想是正确的。

这一过程的伪代码算法描述为：

```

low      : 范围下界
high     : 范围上界
a ← low
repeat:
    if a 为偶数并且 a>2 then
        if a 满足歌德巴赫猜想 then

```



```

        输出一种结论
    else
        设置标志,跳出循环
    endif
endif
a←a+1
until a>high
if 设置标志 then 歌德巴赫猜想不成立
else 在[low,high]内歌德巴赫猜想成立

```

现在问题的核心变为如何验证一个偶数  $a$  是否满足歌德巴赫猜想,即偶数  $a$  能否表示为两个素数之和。可以这样考虑这个问题:

一个正偶数  $a$  一定可以表示成为  $a/2$  种正整数相加的形式。这是因为  $a=1+(a-1)$ ;  $a=2+(a-2)$ ; ……;  $a=a/2-1+a/2+1$ ;  $a=a/2+a/2$ ; 共  $a/2$  种。后面还有  $a/2-1$  种表示形式与前面  $a/2-1$  种表示形式相同,因此可以不考虑。那么,在这  $a/2$  种正整数相加的形式中,只要存在一种形式  $a=i+j$ , 其中  $i$  和  $j$  均为素数,则就可以断定该偶数  $a$  满足歌德巴赫猜想。因此,判断一个大于 2 的偶数  $a$  是否满足歌德巴赫猜想的伪代码算法描述为:

```

i←1
repeat:
    if i 是素数 and a-i 也是素数 then
        设置标志,跳出循环
    endif
i←i+1
until i>a/2
if 设置标志 then a 满足歌德巴赫猜想
else a 不满足歌德巴赫猜想

```

最后可将上述两个算法映射成为两个函数,通过函数的调用实现该功能。

### 程序清单

```

/*****Procedure
Guess*****/
#include <string.h>
#include <stdio.h>

int isGoldbach(int a);
int TestifyGB_Guess(int low,int high);
int isPrime(int i);
void main()
{
    /*验证 1~100 以内的歌德巴赫猜想*/
    printf("Now testify Goldbach Guess in the range of 1~100\n\n");
    if(TestifyGB_Guess(1,100))
        printf("\nIn the range of 1~100,Goldbach Guess is right.\n");
    else printf("\nGoldbach Guess is wrong.\n");
    getchar();
}

int TestifyGB_Guess (int low,int high)
{
    int i,j=0;
    int flag=0;
    for(i=low;i<=high;i++)
        if(i%2==0&&i>2)
            if(isGoldbach(i)){
                j++;
                if(j==5){
                    printf("\n");
                }
            }
}

```



```

        j=0;
    }
    }
    else
        {flag=1;break;}
    if(flag==0)
        return 1;
    else
        return 0;
}

int isGoldbach(int a)
{
    int i,flag=0;
    for(i=1;i<=a/2;i++)
    {
        if(isPrime(i)&& isPrime(a-i))
        {
            flag=1;
            printf("%d=%d+%d ",a,i,a-i);
            break;
        }
    }
    if(flag==1)
        return 1;
    else
        return 0;
}

int isPrime(int i)
{
    int n,flag=1;
    if(1==i)flag=0;
    for(n=2;n<i;n++)
        if(i%n==0){flag=0;break;}
    if(flag==1)
        return 1;
    else
        return 0;
}

```

### 运行效果

```

Now testify Goldbach Guess in the range of 1~100
4=2+2 6=3+3 8=3+5 10=3+7 12=5+7
14=3+11 16=3+13 18=5+13 20=3+17 22=3+19
24=5+19 26=3+23 28=5+23 30=7+23 32=3+29
34=3+31 36=5+31 38=7+31 40=3+37 42=5+37
44=3+41 46=3+43 48=5+43 50=3+47 52=5+47
54=7+47 56=3+53 58=5+53 60=7+53 62=3+59
64=3+61 66=5+61 68=7+61 70=3+67 72=5+67
74=3+71 76=3+73 78=5+73 80=7+73 82=3+79
84=5+79 86=3+83 88=5+83 90=7+83 92=3+89
94=5+89 96=7+89 98=19+79 100=3+97
In the range of 1~100,Goldbach Guess is right.

```

## 17.8 百钱买百鸡问题

### 问题的提出

我国古代数学家张丘建在《算经》一书中曾提出过著名的“百钱买百鸡”问题。该问题叙述如下：鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一；百钱买百鸡，则翁、母、雏各几何？请编写C程序，解决“百钱买百鸡”。

### 问题的分析

如果用数学的方法解决百钱买百鸡问题，可将该问题抽象成如下的方程组：

设鸡翁  $x$  只，鸡母  $y$  只，鸡雏  $z$  只。

则：

$$\begin{cases} 5x+3y+(1/3)z=100 \\ x+y+z=100 \end{cases}$$

显然该方程组应有无数多解，因为根据代数知识，该方程组必有一个自由未知数，因此其解是无数组的。但是作为一个实际问题，其解是有限的，因为这里有约束条件：所有解（ $x, y, z$  取值）必须为正整数。所以，这就把该问题的解集放到了一个有限的空间内来进行讨论。

因此，解决百钱买百鸡问题最为直观的方法就是在（ $x, y, z$ ）的取值空间上穷举出所有可能的取值，只要（ $x, y, z$ ）的取值满足上述两个方程，就一定是百钱买百鸡问题的解。

现在讨论（ $x, y, z$ ）的取值空间。最简单的划分方法就是： $x, y, z \in \mathbb{R}$  且  $0 \leq x \leq 100$ ； $0 \leq y \leq 100$ ； $0 \leq z \leq 100$ 。这是因为无论  $x, y, z$  都不可能超过 100 或小于 0，且  $x, y, z$  均为正整数。因此，百钱买百鸡问题的解空间大小为  $101^3$ ，即要穷举出这  $101^3$  个可能的解，并在这  $101^3$  个可能的解中找出百钱买百鸡问题的真正的解。

下面给出百钱买百鸡问题的伪代码算法描述。

```
i ← 0
j ← 0
k ← 0
repeat:
    j ← 0
    repeat:
        k ← 0
        repeat:
            if i, j, k 的值符合上述两个方程 then
                (i, j, k) 为一个解向量，输出之。
            endif
            k ← k + 1
        until k > 100
        j ← j + 1
    until j > 100
    i ← i + 1
until i > 100
```

上述算法利用了三重循环遍历了整个（ $x, y, z$ ）的取值空间，然后从中找出该问题的解。



## 程序清单

```

/*****Procedure BQBJ*****/
#include <string.h>
#include <stdio.h>

int accord(int i,int j,int k) ;

void main()
{
    int i,j,k;
    printf("The possible plans for buying 100 fowls with 100 yuan are:\n\n");
    for(i=0;i<=100;i++)
        for(j=0;j<=100;j++)
            for(k=0;k<=100;k++)
                if(accord(i,j,k))
                    printf("cock=%d,hen=%d,chicken=%d\n",i,j,k);
    getchar();
}

int accord(int i,int j,int k)
{
    if(5*i+3*j+k/3==100&&k%3==0&&i+j+k==100) /*显然 k 必为 3 的整数倍*/
        return 1; /*符合百千百鸡要求返回 1*/
    else
        return 0; /*不符合百千百鸡要求返回 0*/
}

```

注意：增加约束条件  $k\%3==0$  的原因是  $k$  为整型数据，因此  $k/3$  的结果会自动舍弃小数部分，这样会造成误差（例如  $78/3=26$ ， $80/3=26$ ），影响最终结果。而这里  $k$  必为 3 的整数倍，因此增加约束条件  $k\%3==0$ 。

## 运行效果

```

The possible plans for buying 100 fowls with 100 yuan are:
cock=0,hen=25,chicken=75
cock=4,hen=18,chicken=78
cock=8,hen=11,chicken=81
cock=12,hen=4,chicken=84

```

17.9 求  $\pi$  的近似值

## 问题的提出

编写一个 C 程序，用来求出  $\pi$  的近似值。

## 问题的分析

求  $\pi$  最为常用的方法有两种。

方法 1：利用“正多边形逼近”法求  $\pi$ 。

“正多边形逼近”法求  $\pi$  的核心思想是极限。假设一个直径  $d$  为 1 的圆，只要求出该圆的周长  $C$ ，就可以通过  $\pi=C/d$  的方法求出  $\pi$  的值。所以关键是求出该圆的周长  $C$ 。这里



用“正多边形逼近”的方法求圆的周长。

早在我国古代，数学家们就知道应用正多边形逼近的方法近似地求解圆的周长，称其为“割圆术”。其核心思想是：当一个圆的内接正多边形边数越多时，其边长就越接近外接的圆周长，如图 17-2 所示。

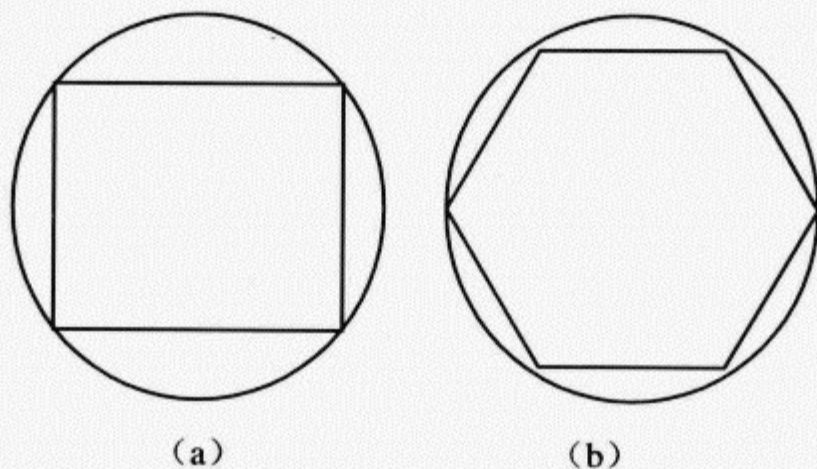


图 17-2 圆的内接正多边形

显然图 (b) 中的内接正六边形的周长较图 (a) 中的内接正四边形的周长更加接近于其外接圆的周长。

现在有这样的迭代关系：设单位圆的内接多边形的边长为  $b$ ，边数为  $i$ ，则多边形边数加倍后，新多边形边长为： $x = \sqrt{2 - 2\sqrt{1 - b^2}}/2$ 。其中  $\sqrt{\quad}$  为开方运算。这样新多边形的周长就为： $C = 2ix$ ，而原先多边形的周长为： $C = bi$ 。

如果最初单位圆的内接多边形为正四边形，其边长为  $\sqrt{2}/2$ ，于是  $b = \sqrt{2}/2$ ， $i = 4$ 。那么边数加倍后，新八边形的边长为  $\sqrt{2 - 2\sqrt{1 - (\sqrt{2}/2)^2}}/2$ ， $i = 8$ 。如此迭代，可求出十六边形、三十二边形的边长等，再乘以相应的边数得到周长  $C$ ，用这个周长  $C$  来近似代替其外接圆的周长，进而求出  $\pi$  的近似值。

利用“正多边形逼近”法求  $\pi$  的伪代码算法如下：

```

n ← 要迭代的次数 (迭代次数越多越精确)
div ← 0
b ← sqrt(2)/2, i ← 4           ; 单位圆内接正四边形的初始化
repeat:
    x ← sqrt(2 - 2sqrt(1 - b^2))/2
    b ← x
    i ← i * 2
    div ← div + 1
until div > n
C = ix
π ≈ C / 1

```

方法 2：应用“蒙特卡洛”方法求  $\pi$ 。

蒙特卡洛方法又叫做随机数方法，它是利用概率论的思想解决实际问题。

利用蒙特卡洛方法求  $\pi$  的核心思想是：在一个边长为  $r$  的正方形中，以一个顶点为圆心， $r$  为半径作一个  $1/4$  圆，随机向该正方形中投入点，其中落入该  $1/4$  圆内的点的概率的 4 倍就是  $\pi$  的近似值，如图 17-3 所示。



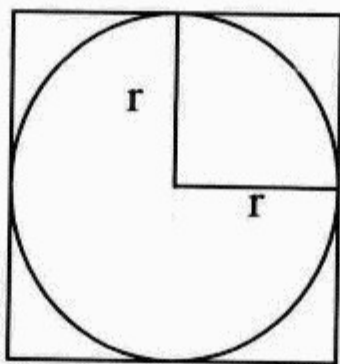


图 17-3 随机点落在 1/4 圆内

根据几何概率的知识, 随机点落入 1/4 圆中概率为 1/4 圆面积与边长为  $r$  的正方形面积的比值, 即  $1/4\pi r^2 : r^2$ , 结果为  $1/4\pi$ 。因此, 此概率的 4 倍就是  $\pi$  的近似值。

应用“蒙特卡洛”方法求  $\pi$  的伪代码算法如下:

```
inCircle ← 0           ; 记录落入 1/4 圆内的点数
count ← 随机点数       ; 向正方形内随机点的数目, 越多越精确
repeat:
    设置随机点的横坐标 x (0~100 内)
    设置随机点的纵坐标 y (0~100 内)
    if  $x^2 + y^2 \leq 10000$  then
        (x,y) 落入半径为 100 的 1/4 圆中, inCircle ← inCircle + 1.
    endif
    count ← count - 1
until count ≤ 0
 $\pi \approx (\text{inCircle} / \text{随机点数}) * 4$ 
```

这里要注意, 圆的半径选取与计算结果无关。

下面给出上述两种方法的程序实现。

### 程序清单

(1) 用“正多边形逼近”法计算  $\pi$  的近似值。

```
/******Procedure PI_1******/
#include <string.h>
#include <stdio.h>
#include <math.h>

double getPI(int n);

void main()
{
    int n;
    double PI;
    printf("Please enter accuracy\n");
    scanf("%d", &n);
    PI = getPI(n);
    printf("The similar value of PI is\n%f\n", PI);
    getchar();
}

double getPI(int n)
{
    int div, i = 4;
    double b = sqrt(2) / 2.0;
    double c = 0.0;
    for (div = 0; div < n; div++)
```

```
{
    b=sqrt(2.0-2.0*sqrt(1.0-b*b))*0.5;
    i=i*2;
}
c=b*i;
return c;
}
```

### 运行效果

迭代3次:

```
Please enter accuracy
3
The similar value of PI is
3.136548
```

迭代6次:

```
Please enter accuracy
6
The similar value of PI is
3.141514
```

迭代11次:

```
Please enter accuracy
11
The similar value of PI is
3.141593
```

从运行结果上可以看出, 迭代次数越多, 所求的 $\pi$ 值越精确。

(2) 应用“蒙特卡洛”方法求 $\pi$ 的近似值。

```
/******Procedure PI_2******/
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

double getPI(int n);

void main()
{
    double PI;
    int n;
    printf("Please enter the number of random points for test\n");
    scanf("%d",&n);
    PI=getPI(n);
    printf("The similar value of PI is\n%f\n",PI);
    getchar();
}

double getPI(int n)
{
    int inCircle=0;
    float x,y;
    int count=n;
    while(count)
    {
        x=random(101);
        y=random(101);
        if(x*x+y*y<=10000)
            inCircle++;
    }
}
```



```

        count--;
    }
    return 4.0*inCircle/n;
}

```

### 运行效果

```

Please enter the number of random points for test
10000
The similar value of PI is
3.135600

```

因为“蒙特卡洛”方法具有随机性，理论上当随机点数目越大时，所求的结果就越逼近  $\pi$  的真实值，但实际计算时，结果精确性并不一定随着随机点数目的增加而一定更加精确。

## 17.10 爱因斯坦的阶梯问题

### 问题的提出

爱因斯坦曾出过这样一道有趣的数学题：有一个长阶梯，若每步上 2 阶，最后剩 1 阶；若每步上 3 阶，最后剩 2 阶；若每步上 5 阶，最后剩 4 阶；若每步上 6 阶，最后剩 5 阶；只有每步上 7 阶，最后刚好一阶也不剩。请问该阶梯至少有多少阶。编写一个 C 程序解决这个问题。

### 问题的分析

把这个问题抽象成数学表达就是求一个满足以下条件的数  $x$ ：

$$\begin{aligned}
 x \bmod 2 &= 1 \\
 x \bmod 3 &= 2 \\
 x \bmod 5 &= 4 \\
 x \bmod 6 &= 5 \\
 x \bmod 7 &= 0
 \end{aligned}$$

显然， $x$  的取值应该有无穷多个，但这里要求取最小的那个解。

从上面这一串表达式中不难发现  $x$  一定是 7 的整数倍，这是因为  $x \bmod 7 = 0$ 。从这一点出发就不难得到解决该问题的一个简单而直观的方法：可以依次递增地求出 7 的整数倍的值 ( $7*1$ 、 $7*2$ 、 $7*3$ ……)，每求出一值，就用该值与 2、3、5、6 进行取模运算，最先得到的满足上述 5 个方程式的  $x$  值即为本题的答案。

解决该问题的伪代码算法如下：

```

x ← 7
repeat
    if (x mod 2=1) and (x mod 3=2) and (x mod 5=4) and (x mod 6=5) then
        x 为所求答案，保存答案
        设置标志，跳出循环
    endif
until 超出循环范围
if 设置了标志 then 输出计算结果

```



```
else 输出在指定范围内没有找到答案
endif
```

### 程序清单

```
/******Procedure Einstein's
question******/
#include <string.h>
#include <stdio.h>

void main()
{
    int x=7,i,res,flag=0;
    for(i=1;i<=100;i++)/*将循环次数定为100*/
    {
        if((x%2==1)&&(x%3==2)&&(x%5==4)&&(x%6==5))
        {
            res=x;
            flag=1;
            break;
        }
        x=7*(i+1);
    }
    if(1==flag)
        printf("The result of Einstein's question is %d",res);
    else
        printf("In this rage cannot get result\n ");
}
```

### 运行效果

```
The result of Einstein's question is 119
```

## 17.11 可扩展的数列排序

### 问题的提出

编写一个C程序，实现这样的功能：从键盘输入任意个整数，以0作为结束标志，对这个整数序列从小到大排序，并输出排序后的结果。

### 问题的分析

在介绍数组时，曾经介绍过用数组存储一组数据，然后对该数组进行“冒泡排序”或者“选择排序”来实现从小到大重新排列这个数组，最后顺序输出数组中的元素值即可。但是本题要求从键盘输入任意个整数，并且以0作为结束标志，因此在数据的组织上就不可能再使用数组了，因为数组的内存分配是在程序编译时完成的，在编写程序代码时必须指定数组的大小。而本程序要求从键盘输入任意个整数，也就是说占用的内存空间大小并不固定，使用数组是难以胜任的，因此必须重新考虑组织数据的数据结构。

其实解决这个问题最简单的方法就是使用链表这种数据结构。因为链表的存储空间是分配在系统的动态存储区的，因此可以在程序执行时动态地分配内存，这样就可以轻松地解决可扩展的数列排序问题了。有关链表的操作等知识可参看第16章的相关介绍，有



关数列排序算法的知识,可参看前面章节有关“冒泡排序”、“选择排序”的算法介绍,这里不再赘述。

### 程序清单

```

/*****Dynamic
Sort*****/
#include <string.h>
#include <stdio.h>
#include <malloc.h>
/*定义链表的节点,有两个域, data 为数据域, next 为指向后继节点的指针域*/
struct Node
{
    int data;
    struct Node *next;
};
/*初始化链表,生成一个节点,为其数据域赋值 d, 返回该节点的指针*/
struct Node * InitList(int d)
{
    struct Node *p;
    p=(struct Node *)malloc(sizeof(struct Node));
    p->data=d;
    p->next=NULL;
    return p;
}

/*插入节点,给定前驱节点指针 p, 在 p 的后面插入一个新的节点,为其数据域赋值 d*/
void InsertNode(struct Node *p,int d)
{
    p->next=(struct Node *)malloc(sizeof(struct Node));
    p->next->next=NULL;
    p->next->data=d;
}

/*基于链表的冒泡排序算法*/
void Sort(struct Node *q)
{
    struct Node *p=q;
    int t,i,j,k=0;
    while(p) {k++; p=p->next;}
    p=q;
    for(i=0;i<k-1;i++)
    {
        for(j=0;j<k-i-1;j++)
        {
            if(p->data>p->next->data)
            {
                t=p->data;
                p->data=p->next->data;
                p->next->data=t;
            }
            p=p->next;
        }
        p=q;
    }
}

/*打印出排序后的新链表中的内容*/
void Print(struct Node *q)
{

```



```
while(q)
{
    printf("%d ",q->data);
    q=q->next;
}

/*主函数*/
main()
{
    int d;
    struct Node *p,*head;
    printf("Please input some integer digit and type 0 for end\n");
    scanf("%d",&d);
    if(d)
        head=p=InitList(d);
    scanf("%d",&d);
    while(d)
    {
        InsertNode(p,d);
        p=p->next;
        scanf("%d",&d);
    }
    Sort(head);
    Print(head);
}
```

运行效果

```
Please input some integer digit and type 0 for end
6 3 3 1 5 12 25 11 8 9 0
The result of sort is
1 3 3 5 6 8 9 11 12 25
```

## 17.12 八皇后问题

### 问题的提出

八皇后问题是一道有趣而经典的数学问题。问题描述为：求解如何在一个  $8 \times 8$  的棋盘上无冲突地摆放 8 个皇后棋子。在国际象棋里，皇后的移动方式为横竖交叉的，因此在任意一个皇后所在位置的水平、竖直、以及  $45^\circ$  斜线上都不能出现皇后的棋子，如图 17-4 所示。

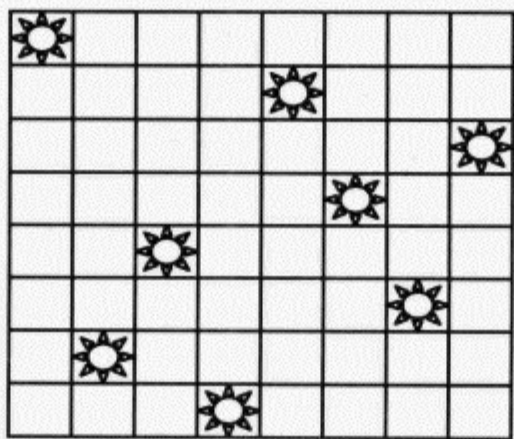


图 17-4 八皇后问题的一种求解



求出所有符合要求的摆放方法（即求出所有的解）。

### 问题的分析

解决八皇后问题的方法有很多，最常见的有回溯法、递归方法、穷举法、概率算法（拉斯韦加斯算法）等。这里介绍经典的递归算法求解八皇后问题。

求解八皇后问题的过程可以直观地描述为如下一个递归的过程。

```
j=1
repeat:
在 n*n 的棋盘上第 1 行第 j 列放置一个皇后；
在剩余的棋盘上（非第 1 行第 j 列以及该皇后 45° 斜线上）构成一个符合要求的 n-1 皇后布局；
j←j+1
until j>n
```

显然，这是一个递归的描述，因为构成一个符合要求的  $n-1$  皇后布局仍然要按照这个算法来进行。在实现上就是递归地调用原函数，重复相同的构造过程，只是参数有所变化。

以上只是简单的伪代码算法描述，旨在说明解决八皇后问题的递归思想，具体的代码实现可参看下面的程序清单。

### 程序清单

```
/******Eight
Queen******/
#include <string.h>
#include <stdio.h>
int count=0; /*计数变量，用来记录八皇后问题的解的个数，它是一个全局变量*/

/******
/*notEqual 函数为 EightQueen 函数所调用 */
/*目的是用来判断棋盘的 row 行第 j 列能否*/
/*摆放一个皇后，如果能够摆放返回 1，否 */
/*则返回 0*/
/******/

int notEqual(int row,int j,int (*chess)[8])
{
    int i,k,flag1=0,flag2=0,flag3=0,flag4=0,flag5=0;
    for(i=0;i<8;i++)
        if(*(*(chess+i)+j)!=0)
        {
            flag1=1;
            break;
        } /*判断列方向*/

    for(i=row,k=j;i>=0 && k>=0;i--,k--)
        if(*(*(chess+i)+k)!=0)
        {
            flag2=1;
            break;
        } /*判断左上方向*/

    for(i=row,k=j;i<8 && k<8;i++,k++)
        if(*(*(chess+i)+k)!=0)
        {
            flag3=1;
            break;
        } /*判断右下方向*/
```



```

    for(i=row,k=j;i>=0 && k<8;i--,k++)
        if(*(*(chess+i)+k)!=0)
        {
            flag4=1;
            break;
        }/*判断右上方向*/
    for(i=row,k=j;i<8 && k>=0;i++,k--)
        if(*(*(chess+i)+k)!=0)
        {
            flag5=1;
            break;
        }/*判断左下方向*/

    if(flag1==1||flag2==1||flag3==1
        ||flag4==1||flag5==1)return 0; /*如果有一个方向不符合要求, 则返回 0, */
        /*表明第 row 行第 j 列不能摆放皇后*/
    else return 1; /*否则返回 1*/
}

/*****
/*EightQueen 函数实现八皇后问题的递归*/
/*求解。当形成符合要求的八皇后棋盘局面*/
/*时, 打印出棋盘的布局, 用一个 0-1 矩阵*/
/*表示棋盘, 0 代表空, 1 代表皇后*/
/*参数: row 表示起始行; n 表示列数等于 8*/
/*(*chess)[8]为指向棋盘每一行的指针*/
*****/

EightQueen(int row,int n,int (*chess)[8])
{
    int j,i;
    int k,t;
    int chess2[8][8];
    for(i=0;i<8;i++)
        for(j=0;j<8;j++)
            chess2[i][j]=*(*(chess+i)+j); /*复制棋盘, 用作递归使用*/
    if(row==8) /*递归结束条件, 形成符合要求的八皇后棋盘局面*/
    {
        for(k=0;k<8;k++){
            for(t=0;t<8;t++){
                printf("%d ",*(*(chess2+k)+t)); /*打印棋盘*/
            }
            printf("\n");
        }
        printf("\n\n");
        getchar();
        count++; /*记录解的个数*/
    }
    else{ /*不符合递归结束条件, 继续进行递归运算*/
        for(j=0;j<n;j++){
            if(notEqual(row,j,chess)) /*判断棋盘的第 row 行第 j 列能否摆放一个皇后*/
            {
                for(i=0;i<8;i++)
                    *(*(chess2+row)+i)=0;
                *(*(chess2+row)+j)=1; /*向棋盘的第 row 行第 j 列摆放皇后*/
                EightQueen (row+1,n,chess2); /*递归地调用 EightQueen 函数*/
            }/*endif*/
        }/*endfor*/
    }/*end else*/
}

```



```

/*主函数*/
main()
{
    int chess[8][8],i,j;
    for(i=0;i<8;i++)
        for(j=0;j<8;j++)
            chess[i][j]=0;                /*初始化棋盘,全部置0*/

    EightQueen(0,8,chess); /*调用 EightQueen 函数,参数: n=0, row=8, */
                           /*chess 为指向棋盘每一行的指针*/
    printf("The number of the answer for eightqueen is\n"); /*输出八皇后问题的
解的个数*/
    printf("%d\n\n",count);
}

```

### 运行效果

由于八皇后问题的全部解的个数为 92 种,因此无法在这里一一展示,只给出第一种解和最后一种解的棋盘布局情况,并显示记录下来的解的个数。

第 1 种解的棋盘布局:

```

1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0

```

第 92 种解的棋盘布局:

```

0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0

```

输出八皇后问题的解的个数:

```

The number of the answer for eightqueen is
92

```

C 语言作为一门最为普及的编程语言之一，在各类考试中都经常考到，特别是在许多公司的面试题中，经常会涉及 C 语言一些基础而细节的知识。本章精选的 100 道常见的 C 语言试题源自一些公司的面试题，计算机等级考试试题，以及其他类型的考试试题，内容涵盖 C 语言的基本知识和容易出题的考点。相信通过本章的学习，读者能够提高应试能力并且巩固已学到的知识。

## 18.1 C 程序设计的基础知识

C 程序设计的基础知识主要包括 C 程序的结构，C 语言的数据类型、运算符及表达式，程序设计的一些规则等内容。这部分知识相对容易，所以经常为人们忽视，但它是进行 C 程序设计的基础，是最根本的知识。因此作为一名合格的程序设计人员，熟练掌握 C 程序设计的基础知识是很必要的。下面通过一些常见的试例题来巩固这部分知识点。

### 例题 18-1

判断 (A)、(B)、(C)、(D) 四个表达式是否正确，若正确，分别写出运算后 a 的值。

`int a = 4;`

(A) `a+=(a++)`; (B) `a+=(++a)`; (C) `(a++)+=a`; (D) `(++a)+=(a++)`;

分析：

本题考查的知识点是有关复合赋值表达式和自增运算符的知识。根据第 2 章介绍的内容知道，在赋值符“=”之前加上其他二目运算符可构成复合赋值符。如+=，-=，\*=，/=等。如果表达式写成 `a+=b`，它等价于 `a=a+b`。另外就是自增运算符“++”，“++”运算符出现位置不同，表达的意思也不相同。`++i` 表示 i 自增 1 后再参与其他运算；`i++` 则表示 i 参与运算后 i 的值再自增 1，即 i 自增 1 的时刻不同。

针对本题，可以看出 (A)、(B)、(C)、(D) 四个表达式都是复合赋值表达式，为了方便判断和理解，做该题时，首先可以将复合赋值表达式转换为一般的赋值表达式，得到如下结果：

(A) `a=a+(a++)`; (B) `a=a+(++a)`; (C) `(a++)=(a++)+a`; (D) `(++a)=(++a)+(a++)`;

这样就可以明显地看出 (C)、(D) 两个表达式都是错误的。因为 C 语言中规定，一个赋值表达式左侧一定是一个变量（包括一般类型的变量、数组成员、结构成员、指针变量等），而决不能是一个表达式。但是 (C)、(D) 两项赋值号左侧都是自增表达式，而不是有效的变量，所以是错误的，编译时不能通过。



而 (A)、(B) 两个表达式是正确的, 因为它们赋值号左侧均为有效变量  $a$ 。根据上面两个等价的一般赋值表达式 (A)  $a=a+(a++)$  和 (B)  $a=a(++a)$  可知, 经过表达式 (A) 的运算后,  $a$  的结果为 9。因为  $a++$  运算最后执行, 而  $a$  的初值为 4, 因此表达式 (A) 先计算  $a=a+a$ , 结果为 8, 再进行  $a$  的自增 1 运算, 最终得 9。经过表达式 (B) 的运算后,  $a$  的结果为 10。因为  $++a$  运算最先执行, 而  $a$  的初值为 4, 因此表达式 (B) 先计算  $++a$  得 5, 再进行  $a=a+a$  运算, 最终得 10。

答案:

(C)、(D) 错误, 因为赋值号左侧不是有效变量; 经过表达式 (A) 运算后  $a$  的值为 9; 经过表达式 (B) 运算后  $a$  的值为 10。

### 例题 18-2

下面一段程序的运行结果是 ( )

```
main()
{
    char a='a',b;
    printf("%c,",++a);
    printf("%c\n",b=a++);
}
```

(A) b, b (B) b, c (C) a, b (D) a, c

分析:

本题依然考查的是自增运算符 “++” 的用法。这里再次说明一下自增运算符 ++ 和自减运算符 — 的用法。

(1)  $++i$  表示  $i$  自增 1 后再参与其他运算;  $--i$  表示  $i$  自减 1 后再参与其他运算。 $i++$  表示  $i$  参与运算后,  $i$  的值再自增 1。 $i--$  表示  $i$  参与运算后,  $i$  的值再自减 1。

(2) 自增运算符 ++ 和自减运算符 — 都只能作用于变量, 而不能作用于常量或表达式。

(3) 自增运算符 ++ 和自减运算符 — 的结合方向是 “自右向左” 的。

针对本题, 字符型变量  $a$  的初值为 'a', 因此执行语句  $\text{printf}("%c", ++a)$  后的结果为: “b,”。这是因为变量  $a$  要先增 1 后再参与其他运算, 也就是说变量  $a$  的值先变为 'b' 后再打印到终端上, 因此本句的运行结果为: “b,”。而执行语句  $\text{printf}("%c\n", b=a++)$  后的结果为: “b”。这是因为变量  $a$  要先赋值给变量  $b$ , 再进行自增 1 运算。也就是说变量  $b$  的值实际上是变量  $a$  没有进行自增运算之前的值, 即 'b'。

答案:

(A)

### 例题 18-3

请写出下列代码的输出内容。

```
#include<stdio.h>
main()
{
    int a,b,c,d;
    a=10;
```



```
b=a++;
c=++a;
d=10*a++;
printf("b,c,d: %d,%d,%d",b,c,d);
return 0;
}
```

分析:

本题考查的知识点是自增运算符的应用以及混合运算的优先次序。程序的第 5 行将 10 赋值给变量 a, 这样此时 a 的值为 10; 程序第 6 行执行语句 b=a++; 即先将 a 赋值给变量 b, a 再进行自增 1 运算, 这样 b 的值为 10, a 的值为 11; 程序第 7 行执行语句 c=++a; 即先将 a 自增 1, 再把它赋值给 c, 这样 a 的值为 12, c 的值也为 12; 程序第 8 行执行语句 d=10\*a++; 即先将 10\*a 的值赋值给变量 d, a 再进行自增运算, 这样 d 的值为 120, a 的值为 13。最终 a, b, c, d 的值分别为: 13, 10, 12, 120。

答案:

b, c, d: 10, 12, 120

例题 18-4

写出以下程序的运行结果。

```
main()
{
    int a=1,b=2,m=0,n=0,k;
    k=(n=b>a)&&(m=a);
    printf("%d,%d",k,m);
}
```

分析:

本题考查的知识点是关系运算符和逻辑运算符, 以及它们运算时的优先次序。根据第 2 章介绍的内容, 各种运算的优先次序如表 18-1 所示。

表 18-1 各种运算的优先次序

运算符	优先级
! 逻辑非运算符	高
算术运算符 (+-*/.....)	↓
关系运算符 (>= <.....)	
&和	
赋值运算符	
	低

当然括号 “()” 的优先级是最高的。因此对于本题来说, 语句 k=(n=b>a)|| (m=a) 的执行过程是: 先分别计算括号中的表达式; 再做逻辑与运算; 最后做赋值运算。在括号中, 依然要按照运算符的优先次序进行计算, 先做 n=b>a 运算, 因为 b>a 为真, 因此关系表达式 b>a 的值为 1, 然后把 1 赋值给变量 n; 再做 m=a 运算, 这是一个赋值运算, 将变量 a 的值赋值给 m, 此时 m 的值就为 1。而赋值表达式 m=a 本身结果也为 1 (因为赋值成功); 最后做逻辑与运算 1&&1, 结果为 1, 并将 1 赋值给变量 k。所以最终 k 的值为 1, m 的值为 1。

答案:

1, 1



**例题 18-5**

若  $x$  和  $y$  都是整型变量,  $x=100$ ,  $y=200$ , 则 `printf("%d",(x,y))` 的输出结果为 ( )

- (A) 200    (B) 100    (C) 100 200    (D) 输出格式符不够, 输出不确定的值

分析:

本题考查的知识点是逗号表达式。根据第 2 章的内容可知, 逗号在 C 语言中的作用主要有两种: 一是作为分隔符使用; 二是作为语句中的运算符使用。本题中的逗号就是作为运算符使用的。这里将若干个表达式用逗号 “,” 连接, 就组成一个逗号表达式。如果一个逗号表达式为: 表达式 1, 表达式 2, …… , 表达式  $n$ , 其运算顺序是: 先求解表达式 1, 再求解表达式 2, …… , 最后求解表达式  $n$ 。整个逗号表达式的运算结果就是表达式  $n$  的值。所以本题中  $(x,y)$  实际上是一个逗号表达式, 其结果为  $y$  的值, 因此输出的也是  $y$  的值。

答案:

- (A)

**例题 18-6**

以下程序的输出结果是 ( )

```
main()
{
    int a=5,b=4,c=6,d;
    printf("%d\n",d=a>b?(a>c?a:c):(B));
}
```

- (A) 5    (B) 4    (C) 6    (D) 不确定

分析:

本题考查的知识点是 C 语言中唯一的一个三目运算符——条件运算符。通过第 3 章的介绍可以知道, 在一定条件下, 条件运算符可以代替条件语句进行条件判断和执行赋值。条件运算符组成条件表达式的一般形式为: 表达式 1? 表达式 2: 表达式 3。其执行过程是: 如果表达式 1 成立, 则执行表达式 2; 否则执行表达式 3。

本题考查的是一个条件表达式的嵌套形式。由于条件运算符的结合方向是自右至左的, 所以首先判断  $a$  是否大于  $c$ 。因为  $a$  的初值为 5,  $c$  的初值为 6, 因此  $a>c$  为假, 于是表达式  $(a>c?a:c)$  返回  $c$  的值 6。然后比较  $a$  和  $b$  的值的值的大小, 因为  $b$  的初值为 4, 所以  $a>b$  为真, 于是返回前面表达式的值, 也就是返回表达式  $(a>c?a:c)$  的值 6。

答案:

- (C)

**例题 18-7**

下列选项中, 不能用作标识符的是 ( )

- (A) `_1234_`    (B) `_1_2`    (C) `float_a`    (D) `1_int`

分析:

本题考查的知识点是 C 语言中的标识符规则。根据第 2 章介绍的内容知道, 标识符 (identifier) 是指用来标识变量名、符号常数名、函数名、数组名、类型名、文件名等的有效字符序列。C 语言中规定标识符只能由字母、数字、下划线三种字符组成。而且第一



个字符只能是字母或下划线。本例题中选项(D)显然不正确,因为它是以数字开头的。

另外值得注意的是,关键字是一类特殊的标识符,又叫做保留字,用来命名C语言程序中的语句、数据类型、变量属性。因此在定义变量、字符常量、函数等时,变量名、常量名、函数名等是不允许与关键字重名的。例如:企图定义变量 `int float; float char;`等都是非法的,企图定义函数 `void float(……){……;}`等也是非法的。

答案:

(D)

#### 例题 18-8

在16位的C编译系统上,若定义了 `long a`,则下列给 `a` 赋值40000的正确语句是( )

(A) `a=20000+20000`

(B) `a=4000*10`

(C) `a=30000+10000`

(D) `a=4000L*10L`

分析:

本题考查的知识点是整型变量的范围以及长整型变量的表示形式。根据第2章介绍的内容知道,ANSI标准规定整型变量 `int` 的范围是-32 768~32 767。因此如果向一个整型变量中赋值的整数大于32 767或小于-32 768则一定发生溢出。本例中虽然变量 `a` 定义为长整型 `long` (长整型的范围是-2 147 483 648~2 147 483 647),但是如果向选项(A)、(B)、(C)那样赋值,其结果为-25 536,显然是不正确的。原因是系统在进行计算时,将参与运算的两个操作数默认为是一般的整数,因此赋值时 `a` 也作为一般的整型变量接收赋给它的值,这样就发生了溢出。但是如果在整型常量后面加上一个字母 `l` 或者 `L`,则系统在运算时会认为是 `long int` 型常量,再进行赋值时, `a` 就会作为长整型变量来接收赋给它的值,这样就不会发生溢出了。

答案:

(D)

#### 例题 18-9

若定义 `x` 和 `y` 为 `double` 类型的变量,则执行完语句: `x=1;y=x+3/2;`后 `y` 的值是( )

(A) 1 (B) 2 (C) 2.0 (D) 2.5

分析:

本题考查的知识点是数据类型的混合运算。在本题中, `x`, `y` 都定义为 `double` 类型的变量,而题目中的 `3/2` 是两个整型的数据进行运算。按照C语言中运算规则,两个整数做整除运算,其结果也一定是整数,因此 `3/2` 的计算结果为1,而不是1.5。然后再与 `x` 相加。`x` 被赋值为1,而变量 `x` 本身是 `double` 类型,所以 `x` 的值为1.0。最后将1.0与1相加得到 `y`,变量 `y` 本身是 `double` 类型,所以 `y` 的值是2.0。

需要补充的一点是,如果希望得到精确的结果,只需将 `3/2` 改写成为 `3.0/2.0`。这是因为此时是两个浮点数在进行整除,按照C语言中运算规则,两个浮点数整除其结果也是一个浮点数。所以 `3.0/2.0` 的结果为1.5。再与1.0相加便可以得到2.5。

答案:

(C)



**例题 18-10**

已知字符 A 的 ASCII 码为十进制数的 65，下面的一段程序的输出结果是什么？

```
main()
{
    char c1,c2;
    c1='A'+5-'3';
    c2='A'+6-'3';
    printf("%d,%c\n",c1,c2);
}
```

**分析：**

本题考查的知识点是字符型变量，字符的 ASCII 码以及字符型变量与整型变量的关系。通过第 2 章的介绍可以知道，在计算机内部，字符型变量占两字节大小，以 ASCII 码的形式存放在变量的内存单元之中，因此字符型变量同样可以参与运算。

本例题中，已知字符 A 的 ASCII 码为 65，因此在计算'A'+5-'3'时，实际上相当于将 A 的 ASCII 码加上 2（因为字符'5'与字符'3'的 ASCII 码差值为 2），这样 c1 的值就为 67，对应的字符为'C'。同理，在计算'A'+6-'3'时，实际上相当于将 A 的 ASCII 码加上 3（因为字符'6'与字符'3'的 ASCII 码差值为 3），这样 c2 的值为 68，对应的字符为'D'。但是本题规定了输出格式，c1 按整型格式输出，c2 按字符格式输出，因此输出结果应为 67 和 D。

**答案：**

67, D

**例题 18-11**

设有如下枚举类型定义：

```
enum language{Basic=3,Assembly,Ada=100,COBOL,Fortran};
```

枚举量 Fortran 的值为（）

(A) 4    (B) 7    (C) 102    (D) 103

**分析：**

本题考查的知识点为枚举类型定义。通过第 2 章的介绍可以知道，枚举元素本身由系统定义了一个表示序号的数值，从 0 开始顺序定义为 0, 1, 2, ...。但也可以像本题这样改变枚举元素的值。根据枚举常量初始化的原则，初始化后枚举常量的值不再是系统默认的 0, 1, 2, ... 的顺序，而是由程序员指定的值。没有初始化的枚举常量值则是它前面枚举常量值依次加 1。例如本题中 Basic 初始化为 3，则 Assembly 的值就为 4；Ada 初始化为 100，则 COBOL 的值就为 101，Fortran 的值为 102。

**答案：**

(C)

**例题 18-12**

下面是一段 C 语言程序：

```
main()
{
    int n;
    float s;
```



```
s=1;
for(n=10;n>1;n--)
    s=s+1/n;
printf("%6.4f\n",s);
}
```

目的是为了计算  $1+1/2+1/3+\dots+1/10$  的值，但是输出的结果并不正确。问该程序的错误出在哪里？

**分析：**

本题是一道寻找程序中的错误的问题，这类题目在面试中很常见，也会有一些难度。本题所要考查的知识点是有关数据类型转换的问题。首先看到程序的第5行 `s=1`，这样赋值是可以的。因为 `s` 被定义成浮点型变量，当把 `1` 赋值给 `s` 时会自动转换成 `1.0` 进行保存。问题出在程序的第7行 `s=s+1/n`。在算术表达式 `s=s+1/n` 中，`1/n` 为两个整数的除法，因此得到的结果仍然为整数，而不是希望得到的浮点数。在本题中，循环中每一次 `1/n` 的计算结果都为 `0`，这样累加起来的结果也自然不会正确。只要将 `1/n` 改写成为 `1.0/n`，结果就正确了。程序中的第8行为输出计算结果 `s` 的语句，其中 `"%6.4f\n"` 为规定的输出格式，并不会影响计算结果。

**答案：**

程序的第7行 `s=s+1/n` 出错，应改为 `s=s+1.0/n`。

### 例题 18-13

编写一个程序，统计输入的一串字符中的空格、制表符、换行符的个数，输入以 `Ctrl+Z` 结尾。

**分析：**

这类编程题在面试中也是常考的。本题考查的重点是字符串的输入输出以及字符的判断。程序设计的关键是如何辨认出从终端输入的字符哪个是空格符，哪个是制表符，哪个是换行符，然后通过不同的变量记录下来每一种字符的个数即可。解决字符的分类问题可以应用字符的 `ASCII` 码进行判断，因为不同的字符对应不同的 `ASCII` 码。通过查表可知空格符的 `ASCII` 码为 `32`，制表符的 `ASCII` 码为 `9`，换行符的 `ASCII` 码为 `10`。可以通过不同的 `ASCII` 码来区分出它们。

**答案：**

下面给出本题的代码清单。

```
#include <string.h>
#include <stdio.h>
main()
{
    char c;
    int space=0,table=0,enter=0;
    printf("Please input a string:\n");
    scanf("%c",&c);
    while(c!=EOF)
    {
        switch (c) {
            case 32:space++;break;
            case 9: table++;break;
            case 10:enter++;break;
            default:break ;
        }
    }
}
```



```
    }
    scanf("%c",&c);
}
printf("The number of space:%d\n",space);
printf("The number of table:%d\n",table);
printf("The number of enter:%d\n",enter);
getchar();
return 0;
}
```

本程序的执行结果为:

```
Please input a string:
main()
{
    int a,b;
    a=1;
    b=2;
}^Z
The number of space:1
The number of table:3
The number of enter:4
```

## 18.2 顺序结构

顺序结构是 C 程序设计中最简单的一种程序结构,但它是其他复杂程序结构的基础,因此有必要熟练掌握。在顺序结构的 C 程序中,最为常用的就是输出函数 `printf` 和输入函数 `scanf`,这两个函数看似简单,但人们使用它们时往往忽视一些细节,不能充分利用它们的功能。在面试中,也经常会考到一些 `printf` 函数和 `scanf` 函数不常使用的功能和细节。本节将通过对一些例题的分析来巩固顺序结构程序设计的要点以及 `printf` 函数和 `scanf` 函数的使用。

### 例题 18-14

下面一段程序执行后输出的结果是 ( )

```
main()
{
    int x=156,y=012;
    printf("%2d,%2d\n",x,y);
}
```

- (A) 15, 01    (B) 56, 12    (C) 156, 10    (D) 156, 12

分析:

本题考查的知识点是整数的表示形式以及函数 `printf` 的输出格式。根据第 2 章的介绍可知整数常量在 C 语言中有三种表示方法:

- (1) 十进制整数:像 1, 2, -25, 0 都是十进制整数。
- (2) 八进制整数:以 0 开头的数为八进制整数表示。如 0123 就表示八进制整数 123, 换算为十进制整数得 83。
- (3) 十六进制整数:以 0x 开头的数为十六进制整数表示。如 0x123 就表示十六进制整数 123, 换算为十进制整数得 291。



因此本题中的  $y=012$  为整数的八进制表示形式，且  $(012)_8=(10)_{10}$ 。

此外，本题还考查了  $d$  格式符的使用。 $d$  格式符用于十进制整数的输入输出，其用法主要包括以下三种：

(1)  $\%d$ ，按整数的实际值输入输出。

(2)  $\%xd$ ， $x$  为指定的输出字段宽度。若输出的整数位数小于  $x$ ，输出时左侧用空格补充；若输出的整数位数大于  $x$ ，则按实际位数输出。

(3)  $\%ld$ ，输入输出长整型数。

综上所述，本题输出结果为 156，10（因为 156 位数大于 2，012 的十进制表示为 10）。

答案：

(C)

### 例题 18-15

下面一段程序执行后输出的结果是 ( )

```
main()
{
    unsigned int a;
    int b=-1;
    a=b;
    printf("%u",a);
}
```

(A) -1    (B) 65 535    (C) 32 767    (D) -32 768

分析：

本题考查的知识点是整数类型的表示范围以及 `printf` 的输出格式。根据第 2 章中介绍可以知道 C 语言规定可以将变量定义为“无符号”类型，其实就是将存储的第一位不作为符号位，而只作为数据位的第一位。一般情况下定义一个整型变量 `int x`， $x$  默认为有符号整型变量，除非像本题这样在类型说明 `int` 前面加上修饰符 `unsigned`，这样变量的范围才不同。在 C 语言中，有符号整型变量的范围是  $-32\,768 \sim 32\,767$ ；无符号整型变量的范围是  $0 \sim 65\,535$ 。

结合本题， $b$  为有符号整型变量，初值为  $-1$ 。在计算机内部它的存储形式为 `1111111111111111`（补码表示）。将它赋值给无符号整型变量  $a$ ，因此第一位的 1 不作为符号位，而是被当作有实际意义的数位。`1111111111111111` 的大小是 65 535。在函数 `printf` 中，输出格式符为 `u`，表示以十进制的无符号整数输出，因此输出结果为 65 535。

答案：

(B)

### 例题 18-16

写出下面一段程序运行后输出的结果。

```
main()
{
    int a=10,b=30;
    printf("%d\n",a,b);
}
```



分析：

  本题考查的知识点是 printf 函数的输出方式。当 printf 函数中缺少一个格式控制符时，也可以理解为要输出的变量的数目多于格式控制符的数目时，对缺少格式控制符的变量不予输出。对于本题来说，因为输出变量为 a，b，而格式控制符只有一个 %d，因此变量 b 不能被输出，而只输出变量 a。这里注意：程序可以通过编译。

答案：

10

例题 18-17

请写出下面一段程序的运行结果。

```
main()
{
    int y=2456;
    printf("y=%d\n",y);
    printf("y=%8o\n",y);
    printf("y=%#8o\n",y);
}
```

注：用 “-” 表示一个空格符。

分析：

  本题考查的知识点是输出格式控制符的使用。在 C 语言中，共有 9 种输出数据的格式控制符，如表 18-2 所示。

表 18-2  输出数据的格式控制符

格式控制符	含义
d	以十进制形式输出带符号整数
o	以八进制形式输出无符号整数
x	以十六进制形式输出无符号整数
u	以十进制形式输出无符号整数
f	以小数形式输出单、双精度浮点数
e	以指数形式输出单、双精度浮点数
g	以%f、%e中较短的输出宽度输出单、双精度浮点数
c	输出字符
s	输出字符串

  另外还有 4 种标志格式符，它们与上述的 9 种格式控制符结合使用，用来控制输出效果。这 4 种标志格式符如表 18-3 所示。

表 18-3  标志格式符

标志格式符	含义
-	结果左对齐，右边补充空格
+	输出正负号
#	对c，s，d，u类无效，对o类在输出时加前缀o
空格	对x类在输出时加前缀0x，对e，g，f类当结果有小数点时才输出小数点



有了上面的介绍，就不难得出本题的答案了：

`printf("y=%d\n",y)`的运行结果为 `y=2456`，这是最一般的输出形式；

`printf("y=%8o\n",y)`的运行结果为 `y----4360`，因为 `o` 表示以八进制形式输出无符号整数，8 为输出字段宽度；

`printf("y=%#8o\n",y)`的运行结果为 `y---04360`，因为 `#` 表示对 `o` 类在输出时加前缀 `o`。

答案：

`y=2456`

`y----4360`

`y---04360`

### 例题 18-18

设定义 `int a,*p=&a;`，以下的 `scanf` 语句中能够正确地为变量 `a` 读入数据的是（ ）

(A) `scanf("%d",p);` (B) `scanf("%d",a);` (C) `scanf("%d",&p);` (D) `scanf("%d",*p);`

分析：

本题考查的知识点是 `scanf` 函数的参数问题。`scanf` 函数的格式为：`scanf(格式控制,地址表列)`，其中格式控制决定输入的数据类型，其功能与 `printf` 函数的格式控制相同。但是需要提出的是，`scanf` 函数没有精度控制，因此类似 `scanf("%2d",&a)` 这样的函数调用是错误的。地址表列由若干个地址（指针）组成，它可以是参数的地址，也可以是字符串数组的首指针。

本题要求向变量 `a` 读入数据，因此地址表列一定是变量 `a` 的指针。这里定义了一个指针变量 `p`，并赋初值为 `&a`，因此 `p` 的内容即为 `a` 的地址（指针）。显然（A）是正确的。

答案：

(A)

### 例题 18-19

有定义语句 `int x,y`，如果要通过 `scanf("%d,%d",&x,&y)` 语句输入两个整数到变量 `x`，`y`（例如两个整数 2 和 3），下面的四组输入形式中错误的是（ ）

(A) `2 3<回车>` (B) `2, 3<回车>` (C) `2, [空格] 3<回车>` (D) `2, <回车>3<回车>`

分析：

本题考查的知识点是 `scanf` 函数的用法。在使用 `scanf` 函数输入数据时，要按照格式控制的要求输入数据。本题中，两个 `%d` 之间有一个逗号“`,`”，因此在输入数据时，该逗号不能省略。所以（A）不正确。至于在输入数据时输入空格符或着回车符，并不会影响输入数据的结果。

答案：

(A)

## 18.3 分支结构

分支结构的程序主要有两种，即 `if` 语句和 `switch` 开关语句。分支语句广泛地应用于程



序设计中，是一类十分重要的语句。分支语句有两点很重要：一是分支语句的判断条件；二是分支语句的嵌套使用。在面试当中，也往往会针对分支语句的这些重点来出题。本节将分析一些分支程序设计的题目。

**例题 18-20**

下面一段程序运行后的输出结果是 ( )

```
main()
{
    int a=3,b=4,c=5,d=2;
    if(a>b)
        if(b>c)
            printf("%d",d++ +1);
        else
            printf("%d",++d+1);
    printf("%d",d);
}
```

(A) 2    (B) 3    (C) 43    (D) 44

**分析：**

本题考查的知识点是 if 语句的嵌套使用以及自增运算符的使用。根据第 3 章中的内容介绍可知，if 语句允许嵌套使用。但必须注意 else 是与第二个 if 配对的。也就是说，第二个 if 与 else 构成一个 if 语句作为第一个 if 的语句，else 永远同与它最近的那个 if 配对。只有先弄清这一点，才能真正地把握程序的分支走向。

就本题而言，先判断 a 是否大于 b，如果 a 大于 b，则执行 if 语句内层的 if 语句，否则执行最后的打印语句。内层嵌套的 if 语句则判断 b 是否大于 c，如果 b 大于 c，则执行第一条打印语句，否则执行第二条打印语句。在这里 a 小于 b，因此程序不执行内层的 if 语句，而是直接执行最后的打印语句 printf("%d",d)，这样就直接打印出 d 的值来。输出 d 的值 2。

**答案：**

(A)

**例题 18-21**

下面一段程序的输出结果为 ( )

```
main()
{
    int i=0,x=0;
    for(;;)
    {
        if(i==3||i==5)continue;
        if(i==6)break;
        i++;
        x+=i;
    }
    printf("%d",x);
}
```

(A) 10    (B) 13    (C) 21    (D) 程序死循环



分析:

本题考查的知识点是 if 语句的判断条件以及 continue 语句和 break 语句的使用。根据第 3 章的内容介绍可以知道, if 语句通过判断给定的条件是否为真来决定执行的分支, 而判断条件一般是关系表达式或逻辑表达式, 当然也可以是一般的表达式或者常量。continue 语句的作用是结束本次循环, 直接执行下一次循环; break 语句的作用则是跳出循环, 直接执行循环之后的语句。

就本题而言, for 语句构成了一个无条件循环的结构, 但当 i 等于 6 时可通过 break 语句跳出循环; 如果 i 等于 3 或 5, 则不执行循环体内的后续语句, 直接进入下一次循环; 跳出整个循环后, 输出 x 的值。

开始时, x, i 的值都为 0, 然后进入循环中, 发现不符合两个 if 语句的判断条件, 然后执行 i++ 和 x=x+i 语句, 这样 i=1, x=1。进入第二次循环之中, 发现 x, i 都不符合两个 if 语句的判断条件, 然后执行 i++ 和 x=x+i 语句, 这样 i=2, x=3。进入第三次循环之中, 发现 x, i 都不符合两个 if 语句的判断条件, 然后执行 i++ 和 x=x+i 语句, 这样 i=3, x=6。进入第四次循环之中, i 满足第一条 if 语句, 于是结束本次循环, 进入下一次循环之中。这样就会发现 i 的值永远等于 3, 因为接下来的每次循环中都只执行到 continue 语句就结束了本次的循环, 所以 i 的值不可能等于 6, 因此程序也将无限循环下去。

答案:

(D)

### 例题 18-22

阅读以下的程序:

```
main()
{
    int x;
    scanf("%d",&x);
    if(x--<5)printf("%d",x);
    else printf("%d",x++);
}
```

程序运行后, 如果从键盘上输入 5, 则输出结果是 ( )

(A) 3 (B) 4 (C) 5 (D) 6

分析:

本题考查的知识点是 if 语句以及自增自减运算符的使用。从第 2 章中的介绍可以知道, x- 的作用是先进行其他运算, 然后 i 进行自减运算; x++ 的作用是先进行其他运算, 然后 i 进行自增运算。所以, 程序运行后如果从键盘上输入 5, 那么在执行 if(x-<5)语句时, 先判断 x 是否小于 5, 再做 x 的自减 1 运算。因此 x-<5 的运算结果为假 (0), 程序执行 else 语句。在执行 printf("%d",x++)语句时, 此时 x 的值为 4, 于是先输出 x 的值 4, 再进行 x 的自增 1 运算。因此输出的结果为 4。

答案:

(B)



**例题 18-23**

下面一段程序的输出结果为 ( )

```
main()
{
    int a=2,b=-1,c=2;
    if(a<b)
    if(b<0)c=0;
    else c++;
    printf("%d",c);
}
```

(A) 0 (B) 1 (C) 2 (D) 3

**分析:**

本题考查的知识点是 if 语句的嵌套使用。在例 18-20 中已经介绍过, if 语句允许嵌套使用, 在 C 语言中规定 else 永远同与它最近的那个 if 配对。就本题而言, else 与第二个 if 构成一个完整的 if-else 语句作为第一个 if 的执行语句。上述 if 语句嵌套等价于:

```
if(a<b){
    if(b<0)c=0;
    else c++;
}
```

所以 printf 语句为程序最后要执行的语句。

程序例中, 最初 a=2, b=-1, c=2, 在判断 a<b 时值为 0, 因此不执行第一个 if 的执行语句, 也就是不执行第二个 if-else 语句, 而是直接执行 printf 语句。因此打印出 c 的结果为 2。

**答案:**

(C)

**例题 18-24**

写出下面一段程序的执行结果。

```
main()
{
    int i;
    for(i=0;i<3;i++)
        switch(i)
        {
            case0: printf("%d",i);
            case2: printf("%d",i);
            default:printf("%d",i);
        }
}
```

**分析:**

本题考查的知识点是 switch 语句的用法。根据第 3 章中的介绍可以知道, switch 语句的一般形式为:

```
switch(表达式){
    case 常量表达式 1: 语句 1;
    case 常量表达式 2: 语句 2;
    ...
    case 常量表达式 n: 语句 n;
    default : 语句 n+1;
```



```
}
```

它表达的意思是计算表达式的值，并与其后的常量表达式值逐个比较，当表达式的值与某个常量表达式的值相等时，就执行它后面的语句，然后就不再进行判断，继续执行后面所有 case 后的语句了。如果表达式的值与所有 case 后的常量表达式均不相同，则执行 default 后的语句。

单纯地按照这种形式使用 switch 语句起不到分支语句的作用，因为这里的“case 常量表达式”在 switch 语句中只是分支的入口，并不是在该处进行条件判断。因此，一旦表达式与 case 后面的常量表达式匹配成功，就从这个入口处执行下去，然后就不再进行判断，继续执行后面所有 case 后的语句了。所以，为了使得 switch 语句真正起到多分支选择语句的目的，一般将 switch 语句与 break 语句配合使用。

本题中就是一个未经改造的 switch 语句。当表达式的值与常量的值匹配上后，就以此为入口，执行后面的所有语句。switch 语句外层有一个循环，i 从 0 循环到 2，共循环 3 次，执行 3 遍 switch 语句。第一次执行 switch 语句，表达式 i 与 case 0 匹配，因此输出 3 个 0；第二次执行 switch 语句，表达式 i 与所有 case 均不匹配，因此执行 default 分支，输出 1 个 1；第三次执行 switch 语句，表达式 i 与 case 2 匹配，因此输出 2 个 2。

答案：

000122

#### 例题 18-25

若有以下定义：float x; float a,b;，则正确的 switch 语句为（）

(A) switch (x)

```
{
    case 1.0 :printf("*\n");
    case 2.0 :printf("**\n");
}
```

(B) switch(x)

```
{
    case 1,2 : printf("*\n");
    case 3: printf("**\n");
}
```

(C) switch(a+b)

```
{
    case 1: printf("*\n");
    case 1+2;printf("**\n");
}
```

(D) switch(a+b);

```
{
    case 1: printf("*\n");
    case 2;printf("**\n");
}
```



分析:

本题考查的知识点是 switch 语句的语法性质。根据第 3 章的介绍可以知道, switch 的参数不能为浮点型数, case 后面必须是整型数或者整型表达式, 而且一个 case 语句只能有一个整型数或者一个整型表达式。根据这些规则来判断上述选项。

(A) 选项中, case 后面是浮点型数, 这不符合 case 后面必须是整型数或者整型表达式的规定, 所以是错误的。

(B) 选项中, case 后面有两个整型数 1, 2, 并用逗号隔开, 这也是不符合语法规则的, 所以是错误的。

(D) 选项中, switch 语句后面不应该加分号 “;”, 所以也是错误的。

因此只有 (C) 选项正确。

答案:

(C)

### 例题 18-26

下面一段程序运行后输出的结果是 ( )

```
main()
{
    int a=15,b=21,m=0;
    switch(a%3)
    {
        case 0: m++;break;
        case 1:m++;
        switch(b%2)
        {
            default: m++;
            case 0: m++;break;
        }
    }
    printf("%d",m);
}
```

(A) 1 (B) 2 (C) 3 (D) 4

分析:

本题考查的知识点是 switch 语句的嵌套调用以及 switch 语句的用法。switch 语句同样可以嵌套调用, 方法同一般的语句一样。本例题中, 将 switch 语句与 break 语句配合使用, 可以实现语句的分支功能。在执行该程序时, 先将 a%3 表达式的值与 case 后面的常量比较, 如果相等, 就沿着该分支执行下去, 直到执行完所有的 case 语句, 或者执行到 break 语句为止。因为本例中使用了 break 语句, 所以程序不会像例题 18-24 那样一直执行到最后一个 case 语句, 而是执行到 break 语句就跳出了, 从而实现了分支的功能。嵌套的 switch 语句的执行过程与外层 switch 语句的执行过程一样。

具体地, 初始时 a=15, b=21, m=0, 因此 a%3 等于 0, 与 case 0 匹配, 所以执行 m++ 语句, 然后跳出整个 switch 语句, 而 case 1 的分支其实是不执行的。因此最后 m 的值为 1。

答案:

(A)



**例题 18-27**

输入3个数  $x$ ,  $y$ ,  $z$ , 编写一个C程序, 从小到大输出这3个数的值。

分析:

对于数目不多的数字排序问题, 并不需要使用复杂的排序算法, 只要用简单的分支语句就可以解决。本题就可以用 if-else 语句实现, 比较的步骤如下:

- (1) 先比较  $x$  和  $y$ , 找出较小的数保存在变量  $\text{min}$  中, 将较大的数保存在变量  $\text{max}$  中;
- (2) 再将  $\text{min}$  与  $z$  比较:

- 1) 若  $z$  小于  $\text{min}$ , 则将  $\text{min}$  保存在变量  $\text{mid}$  中, 并将  $z$  保存在  $\text{min}$  中;
- 2) 否则, 若  $z$  大于  $\text{max}$ , 则将  $\text{max}$  保存在变量  $\text{mid}$  中, 并将  $z$  保存在  $\text{max}$  中;
- 3) 否则, 将  $z$  保存在  $\text{mid}$  中。

通过上述过程, 变量  $\text{min}$ ,  $\text{mid}$ ,  $\text{max}$  中从小到大地存储  $x$ ,  $y$ ,  $z$  的值。

答案:

下面给出本题的代码清单。

```
#include "stdio.h"
main()
{
    int x,y,z,min,max,mid;
    printf("Enter three integer digit:\n");
    scanf("%d %d %d",&x,&y,&z);
    if(x>y)
        {min=y;max=x;}
    else
        {min=x;max=y;}
    if(min>z)
        {mid=min;
         min=z;}
    else if(z>max){
        mid=max;
        max=z;}
    else
        mid=z;
    printf("The order from litter to large is:\n");
    printf("%d,%d,%d",min,mid,max);
}
```

本程序的执行结果为:

```
Enter three integer digit:
1 15 6
The order from litter to large is:
1,6,15
```

**例题 18-28**

给定一个前缀码表如下:

a: 1, b: 01, c: 001

又知有一个 0/1 字符串为: "001011101001011001", 编写一个C程序, 按照给定的前缀码表为该字符串译码。



分析:

作为一种无二义性的编码,前缀码是一类常见的编码方式。哈夫曼编码就是一种前缀码。现要求编写一个程序来为已知的这串 0/1 代码译码。译码的方法很多,现在介绍一种最直观、最简单的译码方式:应用 switch 语句的嵌套进行译码。

因为是前缀码,所以不必考虑译码时的二义性,例如:

a: 001, b:0011, c:1

这就会出现所谓的译码时的二义性,也就是说当翻译到 0011 时无法确定是 ac 还是 b。消除这个问题,就可以根据码表,通过 switch 嵌套语句编写一个直观的译码器了。

下面给出解决该问题的伪代码算法:

```
Repeat:
  Switch (当前 0/1 码)
  Case 1: 输出 a; break;           ——a:1
  Case 0:
    指针指向下一个 0/1 码
    Switch (当前 0/1 码)
    Case 1: 输出 b; break;         ——b:01
    Case 0:
      指针指向下一个 0/1 码
      Switch (当前 0/1 码)
      Case 1: 输出 c; break;       ——c:001
    指针指向下一个 0/1 码
Until 待翻译的码串结束
```

通过上面给出的应用 switch 嵌套语句实现的译码算法,只扫描一遍 0/1 代码串就可以翻译出全部内容,算法的具体结构要依赖于码表的定义。应用 switch 嵌套语句来实现译码算法,优点在于简单直观,很容易理解,且翻译效率较高。缺点在于代码量较大,如果码表庞大,代码量也会随之增大,而且比较机械,缺乏灵活性。这里旨在说明 switch 语句的嵌套使用。

答案:

下面给出本题的代码清单。

```
#include "stdio.h"
void Decode(char *str,int n);
main()
{
    char str[18]="001011101001011001";
    Decode(str,18);
    getchar();
}
void Decode(char *str,int n)
{
    int i=0;
    while(i<n)
    {
        switch(str[i])
        {
            case '1':printf("a");break;
            case '0':
            {
                i++;
                switch(str[i])
                {
```



```
        case '1':printf("b");break;
        case '0':
        {
            i++;
            switch(str[i])
            {
                case '1':printf("c");break;
            }
            break;
        }
    }
    break;
}
i++ ;
}
```

本程序的执行结果为:

```
cbaabcbac
```

## 18.4 循环结构

循环结构是C程序设计中最为重要的一类,也是包括企业面试在内的许多C语言考试的重点。本节将对循环结构程序设计的一些题目进行讨论解析,重点掌握for循环语句、while循环语句、do-while循环语句的用法,同时加深对break语句以及continue语句的理解,掌握它们在循环语句中的作用。

### 例题 18-29

写出下面一段程序的输出结果。

```
main()
{
    int x=0,y=5,z=3;
    while(z-- >0&&++x<5) y=y-1;
    printf("%d,%d,%d\n",x,y,z);
}
```

分析:

本题考查的知识点是while语句的用法以及自增自减运算符的使用。根据第3章的介绍可知,只有当while后面的表达式的真值为假(0)时,循环操作停止,否则一直执行。在本题中,表达式由逻辑与&&运算符联结而成,也就是只有当两端关系式均为真时,该表达式值为真。在这段程序中,初始值x=0, y=5, z=3,每次循环时先要判断z-->0&&++x<5,只有z-->0并且++x<5时才执行循环体语句。第一次循环时,z大于0,因此z-->0为真,z的值随后变为2;x的值变为1,再与5比较,因此++x<5为真,然后进入循环体,执行y=y-1操作。如此循环下去。为了简洁清晰,表18-4中列出每次循环后x, y, z的取值情况。



表 18-4  x, y, z 的取值情况

循环次数	x	y	z
第一次循环后	1	4	2
第二次循环后	2	3	1
第三次循环后	3	2	0
第四次判断后	3	2	-1

因此本程序只执行 3 次循环操作。当第 3 次循环结束后，x，y，z 的值分别为 3，2，0，然后再进行第四次判断。这时  $z > 0$  的真值为假，z 与 0 比较后 z 做自减 1 操作，其值变为 -1。又由于第一个条件不满足使 while 循环中断，所以表达式的后续判断并不执行，循环体内的语句自然也不会执行。因此最终 x，y，z 的值为 3，2，-1。

答案：  
3，2，-1

例题 18-30

有以下程序：

```
main()
{
    int s=0,a=1,n;
    scanf("%d",&n);
    do
    {s+=1;a=a-2;}
    while(a!=n);
    printf("%d\n",s);
}
```

若要使程序的输出结果为 2，则应从键盘给 n 输入的值为（ ）  
(A) -1    (B) -3    (C) -5    (D) 0

分析：

本题考查的知识点是 do-while 语句的用法。do-while 语句与 while 语句类似，只是执行循环体语句的顺序不同。do-while 语句先执行循环体语句，再判断循环条件是否成立，它与 while 语句判断循环条件的时刻不同。这就说明，不论什么条件，循环都会执行一次。就本题而言，先执行语句  $s+=1;a=a-2;$ ，执行完毕后，s 的值为 1，a 的值为 -1，再进行循环条件判断  $a!=n$ 。显然，如果要使程序的输出结果为 2，即 s 的值为 2，只需进行两次循环操作，因为每次循环 s 的值加 1，这样 a 的值就变为 -3（每次循环 a 的值减 2）。当进行第二次循环判断时（也就是决定是否进行第三次循环时），程序应当终止循环，以保证 s 的值为 2，所以 n 的值应为 -3。

答案：  
(B)

例题 18-31

在编写循环判断时，经常有这样一个不成文的规定，即常量写在等于关系式的左侧。例如：



```
while(0==i){.....}
```

而不写成:

```
while(i==0){.....}
```

请简述这种规则写法的好处。

**分析:**

本题考查的是循环判断的书写技巧。作为一名专业的程序设计人员,应该在工作实践中不断积累类似的编程技巧。

从语法上讲,上述两种写法是等价的,而且也是正确的。但是在进行实际的编程时,建议使用第一种书写形式,因为按照第二种书写,一旦程序员误写为 `i=0`,那么这个循环判断就变为永真的,于是造成死循环。如果按照第一种书写格式,一旦程序员误写为 `0=i`,在程序编译时就无法通过。因为一个变量不可能赋值给一个常量,这个赋值语句实际上是非法的。这种书写方法还经常用于分支语句的条件判断,例如:

```
if(3==i){.....}
```

**答案:**

好处是防止程序员误写时程序陷入死循环,便于程序的调试。

### 例题 18-32

分析下面这个程序执行后的错误或者效果。

```
#define MAX 255
main()
{
    unsigned char A[MAX],i;
    for (i=0;i<=MAX;i++)
        A[i]=i;
}
```

**分析:**

本题考查的知识点是 `for` 循环语句的循环判断条件,数组的定义以及无符号字符型变量的范围。本题中,字符常量 `MAX` 定义为 255,数组 `A` 的下标范围为 `0...MAX-1`,因此 `for` 语句的循环判断条件应为 `i<MAX`。如果像程序那样,就会产生数组越界访问,即 `A[255]=255`,这是第一个错误所在。另外,由于 `i` 被定义为无符号字符型变量,因此变量 `i` 的取值范围为 `0~255`。所以当 `i` 的值为 255 时, `i++` 以后 `i` 又为 0 了,而不是希望得到的 256,这样程序就无法结束,陷入死循环之中。

**答案:**

死循环;数组越界访问。

### 例题 18-33

下面一段程序的执行结果是 ( )

```
main()
{
    int x=23;
    do
        {printf("%d",x--);}
}
```



```
while(!x);  
}
```

(A) 321 (B) 23 (C) 不输出任何内容 (D) 死循环

分析:

本题考查的知识点是 do-while 语句的执行。前面提到过, do-while 语句的循环特点是至少执行一次循环。在本题中, 首先执行循环体语句, 即打印出 x 的值 23, 然后 x 做自减 1 运算, 于是 x 等于 22。然后进行循环判断!x, !x 的真值为假, 不满足继续循环的条件, 因此循环终止。所以程序最终的输出结果为 23。

答案:

(B)

#### 例题 18-34

分析下面一段程序实现的功能。

```
main()  
{  
    int i,s=0;  
    for(i=1;i<10;i+=2) s+=i+1;  
    printf("%d\n",s);  
}
```

分析:

本题考查的知识点是 for 循环语句的用法。根据第 3 章的介绍可知, for 循环语句的一般形式为:

for(表达式 1; 表达式 2; 表达式 3) 语句

for 语句的执行过程为:

- (1) 先求解表达式 1。
- (2) 求解表达式 2, 若其值为真 (非 0), 则执行 for 语句中指定的内嵌语句, 然后执行下面第 (3) 步; 若其值为假 (0), 则结束循环, 转到第 (5) 步。
- (3) 求解表达式 3。
- (4) 转回上面第 (2) 步继续执行。
- (5) 循环结束, 执行 for 语句下面的语句。

就本题而言, i 在循环中每次加 2, i 的初值为 1, 上限为 10, 因此共循环 5 次, 并且 i 表示 1~10 内的奇数。s+=i+1 等价于 s=s+i+1, 其作用是累加 1~10 中所有的偶数。因此上面这段程序实现计算 1~10 中偶数之和。

答案:

计算 1~10 中偶数之和。

#### 例题 18-35

为了在屏幕上打印出如下的金字塔图案, 下面的一段程序在横线处应填写什么?



```

      *
    * * *
  * * * * *
* * * * * *

```

```

main()
{
    int i,j;
    for(i=1;i<=4;i++)
        {for(j=1;j<=4-i;j++) printf(" ");
          for(j=1;j<=___;j++) printf(" * ");
          printf("\n");
        }
}

```

分析:

本题考查的知识点是多重 for 循环语句的应用。在实现一些较复杂的问题时，往往要用到多重循环，因此掌握用多重循环编写程序解决实际问题是很重要的。本题要在屏幕上输出如图所示的图案，从给出的程序可以看出最外层循环控制输出\*的行数，内层的第一个循环控制每一行最开始的空格数，第二个循环则是控制\*的输出。从图上看出，每一行输出\*的个数与\*所在的行数有关，其规律是：第*i*行输出 $2*i-1$ 个星号\*，因此横线上应填写 $2*i-1$ 。

答案:

$2*i-1$

#### 例题 18-36

以下程序的执行结果是 ( )

```

main()
{
    int i,n=0;
    for(i=2;i<5;i++)
        {do
            {
                if(i%3) continue;
                n++;
            }while(!i);
            n++;
        }
    printf("n=%d\n",n);
}

```

(A)  $n=5$  (B)  $n=2$  (C)  $n=3$  (D)  $n=4$

分析:

本题考查的知识点是 do-while 语句、for 语句以及 continue 语句的用法。在本题中，for 语句控制了 3 次循环。do-while 语句则是根据 *i* 的值控制循环次数。continue 只作用于 do-while 循环。程序的执行过程如下：

(1)  $i=2$ ，进入 do-while 循环， $2\%3$  等于 2，于是跳出本次循环，*!i* 为 0，do-while 循环停止，执行 *n++* 语句，这时 *n* 等于 1。

(2) *i++* 变为 3，进入 do-while 循环， $3\%3$  等于 0，因此执行 *n++*，*!i* 为 0，do-while 循环停止，再次执行 *n++* 语句，这时 *n* 等于 3。



(3)  $i++$  变为 4, 进入 do-while 循环,  $4\%3$  等于 1, 于是跳出本次循环,  $!i$  为 0, do-while 循环停止, 执行  $n++$  语句, 这时  $n$  等于 4。

(4) for 循环终止。

因此最终  $n$  等于 4。

答案:

(D)

### 例题 18-37

写出下面一段程序的输出结果。

```
main()
{
    int i=0,a=0;
    while(i<20)
    {
        for(;;)
        {
            if((i%10)==0)break;
            else i--;
        }
        i+=11;
        a+=i;
    }
    printf("%d\n",a);
}
```

分析:

本题考查的知识点是 while 循环语句、for 循环语句以及 break 语句的用法。外层 while 循环由变量  $i$  控制, 内层 for 循环是一个无限循环, 由 break 语句负责终止该循环。程序的执行过程如下:

(1)  $i$  的初值为 0, 于是进入 while 循环, 同时进入 for 循环,  $i\%10$  等于 0, 于是跳出 for 循环,  $i=i+11$ ,  $i$  变为 11,  $a=a+i$ ,  $a$  变为 11。

(2)  $i$  小于 20, 于是进入 while 循环, 同时进入 for 循环,  $i\%10$  等于 1, 于是执行  $i--$  操作,  $i$  变为 10。

(3) 再次进入 for 循环,  $i\%10$  等于 0, 于是跳出 for 循环,  $i=i+11$ ,  $i$  变为 21,  $a=a+i$ ,  $a$  变为 32。

(4) while 循环终止。

最终  $a$  的值为 32。

答案:

32

### 例题 18-38

若有如下程序段, 其中  $s$ ,  $a$ ,  $b$ ,  $c$  均已定义为整型变量, 且  $a$ ,  $c$  均已赋值 ( $c>0$ )。

```
s=a;
for(b=1;b<=c;b++)s=s+1;
```



则与上述程序段功能等价的语句是 ( )

- (A)  $s=a+b$     (B)  $s=a+c$     (C)  $s=s+c$     (D)  $s=b+c$

分析:

本题考查的知识点是 for 循环语句。要解答这个问题, 首先必须搞清楚题目中给出的这段代码的功能。本程序中,  $s$  的初值为  $a$ , for 循环的作用是使  $s$  的值在  $a$  的基础上累加, 每次循环累加 1, 共循环  $c$  次, 也就是累加  $c$ 。因此, 上述程序段的功能等价于赋值语句  $s=a+c$ 。

答案:

- (B)

### 例题 18-39

有 1、2、3、4 共 4 个数字, 能组成多少个互不相同且无重复数字的三位数? 都是多少?

分析:

本题考查的是运用多重循环的知识解决实际问题。可以这样考虑这个问题, 应用一个三重循环, 每层循环都从 1 循环到 4。最内层循环作为个位数字, 最外层循环作为百位数字, 这样就可以组成由 1、2、3、4 这 4 个数字组成的所有三位数, 即  $4^3$  个三位数。但这并不符合题目的要求, 因为题目要求能组成互不相同且无重复数字的三位数, 上面的算法只能保证所有的数不重复, 而不能保证每个数无重复数字。因此在最内层的循环还要做一个筛选, 这样就可以按要求得到互不相同且无重复数字的所有三位数了。

答案:

下面给出本题的代码清单。

```
#include "stdio.h"
#include "conio.h"
main()
{
    int i,j,k,res;
    printf("The result is\n");
    for(i=1;i<=4;i++)
        for(j=1;j<=4;j++)
            for (k=1;k<=4;k++)
            {
                if (i!=k&& i!=j&& j!=k) /*确保 i、j、k 三位互不相同*/
                {
                    res=i*100+j*10+k;
                    printf("%d ",res);
                }
            }
    getch();
}
```

本程序的执行结果为:

```
The result is
123 124 132 134 142 143 213 214 231 234 241 243 312 314 321 324 341 342 412 413
421 423 431 432
```

### 例题 18-40

编写一个 C 程序, 计算下面这个式子:

$$3+33+333+\dots+3333333$$



分析:

本题考查的是运用多重循环的知识解决实际问题。可以考虑应用二重循环解决这个问题,外层循环控制累加数字的个数,内层循环控制当前数字的位数。就这个算式而言,外层循环要循环 7 次,内层循环与外层循环的次数有关,第  $i$  次的内层循环循环  $i$  次,也就是说第 1 次内层循环循环 1 次,第 2 次内层循环循环 2 次,……,第  $n$  次内层循环循环  $n$  次。这样通过一个简单的二重循环就可以计算出该式。

答案:

下面给出本题的代码清单。

```
#include "stdio.h"
#include "conio.h"
main()
{
    long x=3,sum=0;
    int i,j;
    for(i=1;i<=7;i++)
    {
        x=3;
        for(j=1;j<i;j++)
            x=x*10+3; /*计算 3, 33, 333……*/
        sum=sum+x; /*累加求和*/
    }
    printf("\n3+33+...+33333333=%ld\n",sum);
    getch();
}
```

本程序的执行结果为:

```
3+33+...+33333333=3703701
```

## 18.5 数组

数组是 C 语言中的构造数据类型的一种,是由基本数据类型按照一定规则组成的数据结构,在程序设计中有着重要的地位,也是许多考试中的重点。本节重点讨论一些有关数组的题目,其中包括一维数组、二维数组、字符数组等相关知识。

### 例题 18-41

下面的程序段  $b$  的值是多少?

```
int a[10]={1,2,3,4,5,6,7,8,9,10},*p=&a[3],b;
b=p[5];
```

分析:

本题考查的知识点是数组元素的地址。根据第 6 章的介绍可知,对于一维数组,数组名是该数组在内存中的首地址,而数组中每个元素的地址可通过取地址符号  $\&$  或者首地址加偏移量的方法得到。

针对本题,定义了整型数组  $a[10]$ ,那么  $a$  就是该数组在内存中的首地址,  $\&a[3]$  或者



写成  $a+3$  就表示数组中第四个元素的地址，而  $a[3]$ ， $*(a+3)$ ， $*\&a[3]$  则都表示数组中第四个元素的值。程序中将  $\&a[3]$  赋值给了指针变量  $p$ （初始化赋值），这样  $p$  就指向了数组  $a$  的第四个元素，也就是 4。而  $b=p[5]$  等价于  $b=*(p+5)$ ，相当于把从  $p$  开始的第五个元素赋值给  $b$ ，也就是把 9 赋值给  $b$ ，所以  $b$  的值是 9。

答案：

9

#### 例题 18-42

以下程序段给数组所有的元素输入数据，横线处应填（）

```
main()
{
    int a[10], i=0;
    while(i<10) scanf("%d", _____);
    .....
}
```

(A)  $a+(i++)$  (B)  $\&a[i+1]$  (C)  $a+i$  (D)  $\&a[++i]$

分析：

本题考查的知识点是数组元素的地址表示。在本题中，希望给数组的每个元素赋值，并且由键盘向数组中输入数据。这里用 `scanf` 语句向数组中输入数据，所以必须给出每个数组元素的地址，最常用的方法是引用数组的下标，用取地址符号  $\&$  实现。但是本题中在给出数组元素地址的同时还要实现变量  $i$  的自增 1 过程，因此 (B) (C) 都不是答案。(D) 也是错误的，因为  $i$  的自增 1 过程先于取地址操作，这样会导致  $a[0]$  元素的地址无法取到以及赋值时最后一个元素地址越界。(A) 是正确答案，(A) 的写法等价于  $\&a[i++]$ 。

答案：

(A)

#### 例题 18-43

假设 `int` 类型变量占 2 字节，其定义有：`int x[10]={0,2,4}`，则数组  $x$  在内存中所占据的字节数是多少？

分析：

本题考查的知识点是一维数组的定义以及初始化。根据第 6 章的介绍可知，一旦定义了一个数组，就会给该数组分配确定大小的空间，C 语言中规定，不允许使用动态数组定义，例如：

```
int i;
char a[i];
```

因为  $i$  是一个不确定的变量。在程序编译时，数组大小的分配取决于数组方括号  $[]$  里的给定的值，这个值必须是确定的。

题目中定义了一个包含 10 个整型元素的一维数组，每个整型元素大小为 2 字节，因此整个数组的大小为  $10*2=20$  字节，与数组的初始化无关。

答案：

20 字节



**例题 18-44**

写出以下程序段的输出结果。

```
main()
{
    int i,k,a[10],p[3];
    k=5;
    for(i=0;i<10;i++)a[i]=i;
    for(i=0;i<3;i++)p[i]=a[i*(i+1)];
    for(i=0;i<3;i++)k+=p[i]*2;
    printf("%d\n",k);
}
```

**分析：**

本题考查的知识点是一维数组的应用以及 for 循环语句的使用。程序段中共有 3 个 for 循环语句，第一个循环给数组 a 赋值，从 a[0]到 a[9]分别赋值为 0 到 9；第二个循环共循环 3 次，分别将 a[0]，a[2]，a[6]赋值给 p[0]，p[1]，p[2]，这样 p[0]，p[1]，p[2]的值分别为 0，2，6；最后将 p[0]，p[1]，p[2]的值分别乘以 2 累加到 k 上，由于 k 的初值为 5，所以 k 的最终结果为  $5+0*2+2*2+2*6=21$ 。

**答案：**

21

**例题 18-45**

下面一段程序运行后输出的结果是 ( )

```
main()
{
    int a[4][4]={{1,2,3,4},{5,6,7,8},{3,9,10,2},{4,2,9,6}};
    int i,s=0;
    for(i=0;i<4;i++)s+=a[i][1];
    printf("%d\n",s);
}
```

(A) 11 (B) 19 (C) 13 (D) 20

**分析：**

本题考查的知识点是二维数组元素的引用和初始化。题目中首先定义了一个 4 行 4 列的二维数组，并对它进行了初始化。然后通过一个循环语句将数组中 a[i][1]的值累加到变量 s 之中。循环变量 i 从 0 循环到 3，因此  $s=a[0][1]+a[1][1]+a[2][1]+a[3][1]$ ，所以  $s=2+6+9+2=19$ 。

**答案：**

(B)

**例题 18-46**

下面一段程序运行后输出的结果是 ( )

```
main()
{
    int b[3][3]={0,1,2,0,1,2,0,1,2},i,j,t=1;
    for(i=0;i<3;i++)
        for(j=i;j<=i;j++) t=t+b[i][b[j][j]];
}
```



```
printf("%d\n",t);  
}
```

(A) 3    (B) 4    (C) 1    (D) 9

分析:

本题考查的知识点是二维数组的初始化以及二维数组元素的引用。根据第6章中的介绍可知,有两种方法对二维数组进行初始化。

(1) 按行分段赋值。例如:

```
int a[2][3]={{1,2,3},{4,5,6}};
```

这种初始化方法将二维数组看成一个矩阵,把每一行的元素写在一个花括号“{}”中。注意,每一行元素的花括号之间要有逗号分隔。

(2) 按行连续赋值。例如:

```
int a[2][3]={1,2,3,4,5,6};
```

这种初始化方法将二维数组看成一个连续的空间,其效果与第一种方法相同。

上一例中对二维数组的初始化采用的是第一种形式,本例中采用第二种形式。

通过一个二重循环将二维数组b中的不同元素累加到变量t上。具体地,当i=0时,  $t=t+b[0][b[0][0]]=1+b[0][0]=1$ ; 当i=1时,  $t=t+b[1][b[1][1]]=1+b[1][1]=1+1=2$ ; 当i=2时,  $t=t+b[2][b[2][2]]=2+b[2][2]=2+2=4$ , 因此最终t的值为4。

答案:

(B)

#### 例题 18-47

以下不能正确定义二维数组的选项是 ( )

(A) `int a[2][2]={{1},{2}};`

(B) `int a[][2]={1,2,3,4};`

(C) `int a[2][2]={{1},2,3};`

(D) `int a[2][]={{1,2},{3,4}};`

分析:

本题考查的知识点是二维数组的定义及初始化。在对二维数组进行初始化时有以下几点需要注意。

(1) 同一维数组类似,在初始化二维数组时可以只对部分元素赋初值,未赋初值的元素自动取0。

例如:

```
int a[3][4]={{1},{2},{3}};
```

这样只对该二维数组的第一行第一列赋值1,第二行第一列赋值2,第三行第一列赋值3,其余元素的位置自动取0。

(2) 如对全部元素赋初值,则第一维的长度可以不给出。

例如:

```
int a[3][3]={1,2,3,4,5,6,7,8,9};
```

等价于



```
int a[][3]={1,2,3,4,5,6,7,8,9};
```

系统会根据数据的个数自动分配空间。但要注意这种赋值方法只限于对全部元素赋初值的情况。

对于本题, (A) 是正确的, 它表示数组有两行, 每一行的第一个元素分别为 1, 2, 其余元素为 0; (B) 是正确的, 它属于上面介绍的第二种情况, 系统会根据大括号内元素的个数自动判断 a 的维数; (C) 也是正确的, 它表示第一行元素是 1 和 0, 其余为第二行元素。只有 (D) 是错误的, 定义数组时第二维的维数不能省略。

答案:

(D)

#### 例题 18-48

以下不能正确进行字符串赋初值的语句是 ( )

- (A) char str[5]= "good! ";      (B) char str[]="good! ";  
(C) char \*str="good! ";      (D) char str[5]={ 'g', 'o', 'o', 'd' };

分析:

本题考查的知识点是字符数组的初始化方法。根据第 6 章中的介绍可知, 字符数组允许在定义时作初始化赋值, 赋初值的方法一般有两种: 一是直接给定义好的字符数组赋值; 二是给定义的字符型指针变量赋值。但要注意两点规则: 当赋值给定义的字符型指针变量时, 可以使用双引号的字符串直接赋值; 当赋值给定义的字符数组时, 只能通过对其元素赋值, 或在定义数组时进行初始化。而且定义的数组空间应足够大, 能存放下整个字符串和结束标志 '\0'。

本题中 (A) 可以在定义时给字符数组赋初值, 但定义的空间不够, 无法存放字符串结束标志 '\0'。(B)、(C)、(D) 赋初值的方法都是可以的。

答案:

(A)

#### 例题 18-49

下面一段程序有什么问题? 对字符串 string 的输出会有什么影响?

```
main()
{
    char string[10], str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1[i] = 'a';
    }
    strcpy( string, str1 );
}
```

分析:

本题考查的知识点是字符串的表示形式。根据第 6 章中的介绍可知, C 语言中规定, 字符串存储的最后一位应以结束标志 '\0' 结尾。只有这样的字符串才是标准的字符串, 任何基于字符串的处理函数 (例如 strcpy, strcmp 等) 都是将 '\0' 作为识别字符串结尾的标志。另外, 如果要以格式符 "%s" 输出字符串, 系统也是以 '\0' 作为识别字符串结尾的标志。所以



在进行字符串赋值时,除非是系统默认的自动添加'\0'的情况(例如字符串的初始化等),一般情况下建议在字符数组的最后添加结束标志'\0'。

本题中,字符数组 str1 的 10 个单元都被赋值为'a',而没有字符串结束标志'\0',然而 strcpy(char \*s1,char \*s2)的工作原理是,扫描 s2 指向的内存,逐个字符赋值到 s1 所指向的内存,直到碰到'\0',因为 str1 结尾没有'\0',具有不确定性,不知道它后面还会赋值什么内容,因此难以保证赋值的正确性,输出时常有乱码。

**答案:**

调用 strcpy 做字符串复制时具有不确定性;输出时常有乱码。

### 例题 18-50

写出以下一段程序的输出结果。

```
main()
{
    char st[20]= "hello\0\t\\\";
    printf("%d %d\n",strlen(st),sizeof(st));
}
```

(A) 9 9    (B) 5 20    (C) 13 20    (D) 20 20

**分析:**

本题考查的知识点是字符串的表示形式。在上一例题中已经讲到,任何基于字符串的处理函数(例如 strcpy, strcmp 等)都是将'\0'作为识别字符串结尾的标志。strlen 函数是用于测试字符串的长度,它的返回值是字符串的长度,不包括最后的结束符'\0',而函数 sizeof 则是用于测试变量或数组的内存分配的大小,它不属于字符串函数。

因此本题中 strlen(st)的值应为 5,因为结束符'\0'前面仅有 5 个字符,系统默认的字符串长度为 5。而 sizeof(st)的值为 20,因为定义的字符数组 st 占据 20 个单元(字节)空间。

**答案:**

(B)

### 例题 18-51

如果输入 ABC,当执行下面的程序时,输出结果是()

```
#include<stdio.h>
#include<string.h>
main()
{
    char str[10]="1,2,3,4,5";
    gets(str);
    strcat(str, "6789");
    printf("%s\n",str);
}
```

(A) ABC6789    (B) ABC67    (C) 12345ABC6    (D) ABC456789

**分析:**

本题考查的知识点是字符串库函数 strlen 的应用以及%s 格式符输出字符串。程序中定义了一个字符数组 str[10],并赋初值。当调用 gets 函数从终端接收一个字符串时,将字符串"ABC"赋值给字符数组 str,并自动在后面加上字符串结束标志'\0'。调用函数 strcat 时,



进行两个字符串的连接。strcat 的执行过程是找到第一个字符串的结束标志'\0'，去掉该结束标志，把要连接的字符串连接上，并在最后添加'\0'。这样字符数组中原有的 12345 就被覆盖掉了，字符数组 str 中存放的元素是"ABC6789\0"。当用格式符"%s"输出字符串时，系统也是以'\0'作为识别字符串结尾的标志，因此本程序输出的结果为 ABC6789。

答案：

(A)

### 例题 18-52

写出下面一段程序的运行结果。

```
main()
{
    char w[][10]={{A,B,C,D},{E,F,G,H},{I,J,K,L},{M,N,O,P}},k;
    for(k=1;k<3;k++)
        printf("%s\n",&w[k][k]);
}
```

分析：

本题考查的知识点是二维数组的初始化。根据第 6 章的介绍可知，有两种方法对二维数组进行初始化：按行分段赋值和按行连续赋值。

就本题而言，数组初始化时第一维的长度没有给出，说明是对全部元素赋值。赋值的方式是第一种，因此数组 w 为一个 4 行 10 列的矩阵。同时，在初始化二维数组时有部分元素没有给出，未赋初值的元素自动取 0。因此该数组的形式如下：

A,B,C,D,0,0,0,0,0,0

E,F,G,H,0,0,0,0,0,0

I,J,K,L,0,0,0,0,0,0

M,N,O,P,0,0,0,0,0,0

在输出时应用了一个循环语句，相当于 printf("%s",&w[1][1]);和 printf("%s",&w[2][2]);，注意这是应用格式符"%s"输出字符串的方法，因此输出的是以 w[1][1]和 w[2][2]为首的两个字符串。也就是 FGH 和 KL。

答案：

FGH

KL

### 例题 18-53

有一个有序的字母序列：a, b, d, f, h, j, l, p, t；编写一个程序，要求从终端输入一个字母，将该字母插入这个序列中使得字母序列依然保持有序，然后输出新的字母序列。

注意：如果输入的字母在原序列中存在，就将该字母插入到已存在字母的后面。

分析：

首先要考虑的是这个字符串的存储方法。要在程序中添加一个新的字母，因此原有的字符数组的容量必须足够大。将新的字母序列输出时，为了方便操作，可将该字符数组按



照字符串形式输出，所以数组最后应该加上字符串结束标志。

然后就要考虑程序的算法。可以通过比较字符的 ASCII 码判断输入的字母应该插入的位置。确定好插入的位置后，将该位置以后的所有字母顺序后移一个单元，再将该字母插入即可。

输出的时候可以应用格式符 “%s” 按照字符串的形式输出。

**答案：**

下面给出本题的代码清单。

```
#include "stdio.h"
main()
{
    char str[11]={'a','b','d','f','h','j','l','p','t'},c;
    int i,j;
    printf("Please input a alpha\n");
    scanf("%c",&c);
    for(i=0;i<9;i++)
        if(str[i]>=c)break;
    if(i<9)
        for(j=8;j>=i;j--)
            str[j+1]=str[j];
    str[i]=c;
    printf("The result is\n");
    printf("%s",str);
    getche();
}
```

本程序的执行结果为：

```
Please input a alpha
m
The result is
abdfhjlmpt
```

```
Please input a alpha
b
The result is
abbdfhjlp
```

```
Please input a alpha
z
The result is
abdfhjlpzt
```

#### 例题 18-54

输入 5 个国家的名称并按字母顺序排列输出。

**分析：**

首先考虑数据的存储问题。可以用字符型的二维数组存储国家名，每一行存储一个国家名，共存储 5 行。因此需要定义一个 5 行 n 列的二维数组，n 要足够大，以便可以放下国家的名字。C 语言规定可以把一个二维数组当成多个一维数组处理。因此本题可以按 5 个一维数组处理，而每一个一维数组就是一个国家名字符串。

然后考虑算法。可以用字符串比较函数 strcmp 对字符串进行大小的比较，然后根据比较的结果对 5 个字符串排序输出。



答案:

下面给出本题的代码清单。

```
#include "stdio.h"
main()
{
    char st[20],cs[5][20];
    int i,j,p;
    printf("input country's name:\n");
    for(i=0;i<5;i++)
        gets(cs[i]);
    printf("\n");
    for(i=0;i<5;i++)
    {
        p=i;
        strcpy(st,cs[i]);
        for(j=i+1;j<5;j++)
            if(strcmp(cs[j],st)<0)
                {p=j;strcpy(st,cs[j]);} /*找到字母小的字符串*/
        if(p!=i)
        {
            strcpy(st,cs[i]); /*实现字符串的调换*/
            strcpy(cs[i],cs[p]);
            strcpy(cs[p],st);
        }
        puts(cs[i]);
    }
    printf("\n");
    getch();
}
```

本程序的执行结果为:

```
input country's name:
China
Japan
America
UK
Russia

America
China
Japan
Russia
UK
```

## 18.6 指针

指针是 C 语言中最为重要的概念之一,也是 C 语言的一个重要特色。通过指针的操作可以灵活地控制变量,有效地表达复杂的数据结构,本节将就指针方面的题目加以归纳解析。

### 例题 18-55

以下程序的运行结果为 ( )

```
main()
{
```



```
int a=7,b=8,*p,*q,*r;
p=&a;
q=&b;
r=p;
p=q;
q=r;
printf("%d,%d,%d,%d\n",*p,*q,a,b);
}
```

(A) 8, 7, 8, 7 (B) 7, 8, 7, 8 (C) 8, 7, 7, 8 (D) 7, 8, 8, 7

分析:

本题考查的知识点是指针变量赋值用法。根据第6章的介绍可知,指针变量是一类用来存放变量地址的变量。如果定义了整型的指针变量  $p$ ,那么  $p$  就可以存放一个整型变量的地址。例如: `int *p,q; p=&q;` 表明  $p$  是一个指向整型的指针变量,  $q$  是一个整型变量,通过赋值语句 `p=&q` 将  $q$  的地址赋值给  $p$ ,也就是说指针变量  $p$  指向了变量  $q$ 。

就本题而言,  $p, q, r$  都定义为指向整型的指针型变量,  $a, b$  为整型变量,并且初始化为 7 和 8,然后用  $p$  指向  $a$ ,  $q$  指向  $b$ ,再将  $p$  和  $q$  中的内容(也就是变量  $a$  和  $b$  的地址)调换,这样  $p$  就指向了变量  $b$ ,  $q$  就指向了变量  $a$ 。最后将  $p$  指向的内容  $*p$ ,  $q$  指向的内容  $*q$ ,以及  $a$  和  $b$  依次打印出来,结果为 8, 7, 7, 8。

答案:

(C)

#### 例题 18-56

设有定义: `int n=0,*p=&n,**q=&p;`,则以下选项中,正确的赋值语句是()

(A) `p=1;` (B) `*q=2;` (C) `q=p;` (D) `*p=5;`

分析:

本题考查的知识点是指针变量的赋值以及指向指针的指针。题目中定义了  $n$  为整型变量,并赋初值为 0,  $p$  为指向整型的指针变量,并赋初值为  $n$  的地址 `&n`,  $q$  为指向指针型变量的指针变量,并赋初值为 `&p`,也就是  $q$  指向  $p$ ,  $p$  指向  $n$ ,  $n$  存放整数 0。

只要搞清楚这个关系就不难解决这道题目。(A) 显然错误,因为  $p$  是指针型变量,它只能存放其他变量的地址;(B) 不正确,因为  $q$  是指向指针的指针变量,它指向的内容  $*q$  一定是一个变量的地址;(C) 也不正确,只能是 `q=&p`,因为  $q$  中只能存放指针变量  $p$  的地址;(D) 是正确的,  $p$  是一个指向整型变量的指针变量,  $p$  指向的内容  $*p$  一定是一个整数。

答案:

(D)

#### 例题 18-57

写出以下一段程序的输出结果。

```
#include<stdio.h>
f(char *s)
{
    char *p=s;
    while(*p!='\0')p++;
    return(p-s);
}
main()
```



```
{
    printf("%d\n",f("ABCDEF"));
}
```

分析:

本题考查的知识点是指向字符串的指针。在本题中,主函数调用了一个函数 f,参数是一个字符串,表示该字符串的首地址。在函数 f 中,将形参 s 赋值给指向字符型的指针变量 p,于是指针变量 p 就指向了字符串"ABCDEF"。然后做一个循环操作,目的是使指针 p 指向字符串的结束标志'\0',最后返回 p-s,实际上就是该字符串的长度 6。

答案:

6

### 例题 18-58

有下面一段程序:

```
#include<stdlib.h>
main()
{
    char *p,*q;
    p=(char *)malloc(sizeof(char)*20);
    q=p;
    scanf("%s%s",p,q);
    printf("%s%s\n",p,q);
}
```

如果从键盘上输入:abc def<回车>,则输出的结果为 ( )

(A) def def (B) abc def (C) abcd (D) d d

分析:

本题考查的知识点是指向字符串的指针以及动态分配内存函数 malloc。在本题中,通过 malloc 函数开辟了一段 20 个字符的数据空间,然后将这段空间的地址赋值给指针变量 p (通过指针类型的强制转换),再将 p 赋值给 q,这样指针变量 p 和 q 指向内存中的同一段数据空间。在执行输入语句 scanf 时,先向 p 指向的数据空间输入字符串"abc",以空格作为结束标志,再向 q 指向的数据空间输入字符串"def",但是由于 p 和 q 指向内存中的同一段数据空间,因此第一次输入的字符串"abc"实际上被第二次输入的字符串"def"覆盖掉了。因此,最终 p 和 q 指向的字符串中只有 def。

答案:

(A)

### 例题 18-59

若有以下说明和语句, int c[4][5],(\*p)[5]; p=c;能正确引用 c 数组元素的是 ( )

(A) p+1 (B) \*(p+3) (C) \*(p+1)+3 (D) \*(p[0]+2)

分析:

本题考查的知识点是指针与数组的关系。首先必须明确 int (\*p)[5]定义的意思。根据第 6 章的介绍可以知道,如果把二维数组 a[3][4]按行分解为一维数组 a[0]、a[1]、a[2]之后,设 p 为指向二维数组的指针变量,可定义为:



```
int (*p)[4]
```

它表示  $p$  是一个指针变量，它指向包含 4 个元素的一维数组，即二维数组的每一行。 $p+i$  表示指向二维数组的第  $i+1$  行。例如： $p+1$  就表示指向二维数组第二行的指针。

本题就是这样， $c$  是一个 4 行 5 列的整型数组，定义的  $(*p)[5]$  表明  $p$  是一个指针变量，它指向包含 5 个元素的一维数组，也就是二维数组  $c[4][5]$  的每一行。把  $c$  赋值给  $p$ ，就等于使  $p$  指向了数组  $c[4][5]$  的第一行。因此，要引用数组  $c$  的元素，(A) 是错误的，因为它只表示二维数组  $c[4][5]$  第二行的首地址；(B) 也是错误的， $*(p+3)$  与  $p+3$  的值是一样的，都表示数组  $c[4][5]$  第四行的首地址；(C) 也是错误的，它表示  $c[1][3]$  的地址；只有 (D) 是正确的，它等价于  $*(*(p+0)+2)$ ，表示数组元素  $c[0][2]$  的值。

答案：

(D)

### 例题 18-60

写出以下一段程序的运行结果。

```
#include<stdio.h>
main()
{
    int a[]={1,2,3,4,5,6,7,8,9,10,11,12},*p=a+5, *q=NULL;
    *q=*(p+5);
    printf("%d,%d\n",*p,*q);
}
```

分析：

本题考查的知识点是指针变量的赋值。题目中，首先将  $p$  指向  $a[5]$ ，将  $q$  指向空，然后将  $*(p+5)$  也就是  $a[10]$  的内容赋值给  $q$  指向的内容。然而  $q$  被定义为空指针，程序也没有对  $q$  进行赋值，因此系统输出的时候，先输出 6 11，然后出现  $q$  没有赋值的错误提示。

答案：

先输出 6 11，然后出现  $q$  没有赋值的错误提示。

### 例题 18-61

分析以下代码并找出问题。

```
main()
{
    char a;
    char *str=&a;
    strcpy(str,"hello");
    printf(str);
    return 0;
}
```

分析：

这类寻找程序漏洞的问题在 C 语言的面试题中经常出现，特别是指针这一部分。因为指针的操作非常灵活，一旦操作不慎，可能会导致程序的崩溃，因此这类题目应当特别注意。

本题中定义  $a$  为一个字符型变量，定义  $str$  为指向字符型的指针变量，然后将  $a$  的地址赋值给  $str$ ，也就是说此时指针变量  $str$  指向变量  $a$ ，然后企图通过字符串拷贝函数，将字符



串"hello"复制给 str 指向的空间。然而 str 只是指向一个字符型变量 a，它并没有指向一个有足够大的空间区域来存放字符串，因此导致程序崩溃。

**答案：**

没有为 str 分配足够的内存空间，将会发生异常。

#### 例题 18-62

分析以下代码并找出问题。

```
#include <stdio.h>
#include <stdlib.h>
void getmemory(char *p)
{
    p=(char *) malloc(100);
    strcpy(p,"hello world");
}
main( )
{
    char *str=NULL;
    getmemory(str);
    printf("%s/n",str);
    free(str);
    getche();
    return 0;
}
```

**分析：**

本题考查的知识点是指针作为函数的参数。在主函数中，定义了 str 为指向字符型的指针变量，并赋初值为 NULL，然后作为参数传递给函数 getmemory，函数 getmemory 的作用是开辟一个 100 字节大小的内存空间，并将字符串"hello world"复制到这个内存空间中去。程序的原意是通过函数 getmemory 将 str 指向开辟的内存空间，然后通过语句 printf("%s/n",str) 显示该字符串，最后再释放掉 str 指向的内存空间。但是在调用函数 getmemory 时发生了错误，因为函数的参数传递都是值传递，被调函数 getmemory 的形参实际上只是对实参的一个拷贝，在函数 getmemory 中所做的一切操作都不会影响到指针变量 str，因此通过语句 printf("%s/n",str) 显示不出字符串"hello world"，而应用 free 函数释放一个空指针 str 是很危险的。

**答案：**

通过语句 printf("%s/n",str) 显示不出字符串"hello world"，而应用 free 函数释放一个空指针 str 是很危险的。

#### 例题 18-63

分析以下代码并找出问题。

```
main()
{
    char *a;
    scanf("%s",a);
}
```



分析:

本题考查的是在应用指针编写程序时应当遵循的一些规则。在本题中,定义了一个指向字符型的指针变量 *a*,然后通过 `scanf` 语句直接向 *a* 所指的空间输入字符串。从语法上讲,这段程序是没有问题的,但是在编程时不提倡这样用。因为 *a* 中的内容是不确定的,即指针变量指向的内存空间的位置是不确定的,它不像数组的数组名那样有明确的指向(指向系统为该数组开辟的内存空间的首单元)。如果指针变量 *a* 指向的是内存中的代码段,那么再向 *a* 所指向的内存空间存储一段数据就会破坏代码段中的程序,影响系统的正常运行。

答案:

不提倡这样用,因为一旦指针变量 *a* 指向的是内存中的代码段,就可能会破坏代码段中的程序,甚至会影响系统的正常运行。

#### 例题 18-64

以下程序运行后的输出结果是 ( )

```
main()
{
    int a[3][3],*p,i;
    p=&a[0][0];
    for(i=0;i<9;i++)p[i]=i+1;
    printf("%d\n",a[1][2]);
}
```

- (A) 3    (B) 6    (C) 9    (D) 2

分析:

本题考查的知识点是数组指针变量。二维数组在内存中的排列规律是按行存放的,因此当 *p* 指向数组的第一个元素时,实际上也就是指向了整个二维数组的第一个元素。*p[i]*等价于 $*(p+i)$ ,表示从起始位置开始的第 *i* 个元素,*p[3]*相当于 *a[1][0]*,*p[4]*相当于 *a[1][1]*,……。因此程序中循环语句的作用是给数组 *a[3][3]* 从 1 到 9 按行赋值。因此 *a[1][2]* 相当于 *p[5]*,其值为 6。

答案:

(B)

#### 例题 18-65

下面一段程序的运行结果是什么?

```
main()
{
    char *ps="this is a book";
    int n=10;
    ps=ps+n;
    printf("%s\n",ps);
}
```

分析:

本题考查的知识点是指向字符串的指针。本题中定义了一个字符型指针变量 *ps*,并为该变量赋初值,初始化了一个字符串。因此该字符串的首地址就赋值给了指针变量 *ps*,*ps* 指向 "this is a book" 这个字符串,接下来将 *ps* 加 10。在 C 语言中,指针变量的加减并不是



真实值上的加减，而是逻辑上的加减。就像本题这样，ps 加 10 并非 ps 的值本身加 10，而是将指针 ps 向后移 10 个单元，单元的大小由每个存储块的大小决定，本题是 1 字节。因此，ps 加 10 后，指针 ps 指向字符 b。最后用格式符%s 输出 ps 指向的字符串，所以输出的结果为 book。（字符串初始化时，系统已经为字符串的最后添加了字符串结束标志'\0'）

答案：

book

### 例题 18-66

下面一段程序的输出结果为（）

```
#include<stdio.h>
#include<string.h>
main()
{
    char b1[8]= "abcdefg",b2[8],*pb=b1+3;
    while(--pb>=b1) strcpy(b2,pb);
    printf("%d\n",strlen(b2));
}
```

(A) 8 (B) 3 (C) 1 (D) 7

分析：

本题有一定难度，它考查的知识点包括指向字符串的指针以及字符串库函数的相关知识。本题首先定义了一个字符数组 b1，并为其初始化字符串"abcdefg"，然后定义了一个字符型指针变量 pb，并将它指向数组 b1 的第 4 个元素 d。然后进行一个循环操作，循环判断条件是一pb>=b1，也就是先做 pb 的自减 1 运算，使 pb 指向前一个字符，然后再判断 pb 的值是否大于等于 b1 的值。如果 pb 的值大于等于 b1 的值，也就是指针 pb 还没有退回到字符串的首地址处，则进入循环体进行字符串的复制操作。那么，最后一轮的字符串复制操作一定是将 b1 的整个字符串复制给 b2，因为最后 pb 一定是等于 b1 的。因此，最终字符串 b2 的长度与字符串 b1 的长度相同，都为 7。

答案：

(D)

### 例题 18-67

写出下面一段程序运行后的输出结果。

```
main()
{
    char *p1,*p2,str[50]= "xyz";
    p1="abcd";
    p2="ABCD";
    strcpy(str+2,strcat(p1+2,p2+1));
    printf("%s",str);
}
```

分析：

本题考查的知识点是指向字符串的指针以及字符串处理函数。p1, p2 为定义的指向字符型的指针变量，p1 指向字符串"abcd"，p2 指向字符串"ABCD"。str[50]为定义的字符指针，



初始化赋值字符串"xyz"。然后进行 `strcpy(str+2, strcat(p1+2, p2+1))` 操作。首先将字符串 `p1+2` 与字符串 `p2+1` 连接, 形成字符串"cdBCD", 然后将该字符串复制到字符串 `str+2` 上, 因此字符串 `str` 的内容变为"xycdBCD", 输出的结果也就是"xycdBCD"。

答案:

xycdBCD

### 例题 18-68

写出下面一段程序的运行结果。

```
struct s
{int x,y;}
data[2]={10,100,20,200};
main()
{
    struct s *p=data;
    printf("%d\n", ++(p->x));
}
```

分析:

本题考查的知识点是指向结构体的指针。定义了 `data[2]` 为一个 `struct s` 类型的结构体数组, `p` 为指向结构体类型的指针, 并将结构体数组的首地址 `data` 赋值给 `p`, 于是 `p` 指向了结构体数组 `data`。然后打印出 `++(p->x)`。`++(p->x)` 表示 `p` 指向的结构体元素的 `x` 域的值加 1, 它等价于 `++data[0].x`。已知 `data[0].x` 的值为 10, 所以做自增运算后输出的结果为 11。

答案:

11

### 例题 18-69

已知定义的整型变量 `a` 和 `b` 初始化为 `a=2, b=3`, 编写一个函数 `swap`, 将变量 `a, b` 的内容交换, 使得 `a` 中存储 3, `b` 中存储 2。

分析:

这是一个典型的应用变量的指针作为函数参数的问题。在 C 语言中, 函数的参数传递遵循值传递的原则, 因此如果要改动主调函数中变量的值, 仅传递变量是不起作用的, 必须使用传递变量的指针(地址)才能改变主调函数中变量的内容。在本题中就是将变量 `a, b` 的地址作为函数的参数进行传递。

答案:

下面给出本题的代码清单。

```
#include <string.h>
#include <stdio.h>
swap(int *a, int *b)
{
    int t;
    t=*b;
    *b=*a;
    *a=t;
}

main()
{
```



```
int a=2,b=3;
printf("Before swap:%d,%d\n",a,b);
swap(&a,&b);
printf("After swap:%d,%d\n",a,b);
getche();
}
```

本程序的执行结果为:

```
Before swap:2,3
After swap:3,2
```

### 例题 18-70

有两个 3\*4 的矩阵:

$$\begin{pmatrix} 1 & 3 & 5 & 2 \\ 2 & 6 & 2 & 1 \\ 3 & 0 & 8 & 9 \end{pmatrix} \quad \begin{pmatrix} 5 & 6 & 8 & 9 \\ 2 & 1 & 0 & 0 \\ 6 & 9 & 2 & 3 \end{pmatrix}$$

编写一个程序 MatrixAdd, 实现两个矩阵的求和。要求: 通过该函数将矩阵的和带回, 在主函数中打印出和矩阵。

分析:

本题考查的知识点是二维数组的指针。有两种方法表示二维数组的指针。一种方法是将二维数组看成是按行存储的一维数组, 这样在传递参数时将数组名作为参数传递, 把该数组名看成是这个一维数组的首地址。例如: 二维数组  $a[2][3]$ , 可以看成是一个一维数组, 其存储形式为:  $a[0][0]$ ,  $a[0][1]$ ,  $a[0][2]$ ,  $a[1][0]$ ,  $a[1][1]$ ,  $a[1][2]$ 。a 是这个一维数组的首地址。那么  $a[3]=*(a+3)$  就表示  $a[1][0]$  的值。这样只要通过一个数组名就可以控制整个二维数组的元素。第二种方法是应用指向二维数组每一行的指针, 例如定义一个指针变量  $\text{int} (*p)[4]$ , 再定义一个二维数组  $a[3][4]$ , 则 p 是一个指针变量, 它指向包含 4 个元素的一维数组, 也就是说 p 可以指向  $a[3][4]$  的每一行, 那么通过指针 p 可以控制整个二维数组的元素。下面根据这两种方法, 给出两种解决方案。

答案:

第一种方法: 将二维数组看成是按行存储的一维数组。

```
#include <string.h>
#include <stdio.h>
MatrixAdd(int *a,int *b,int *r)
{
    int i;
    for(i=0;i<12;i++)
        *(r+i)=*(a+i)+*(b+i); /*将 a,b,r 都看成是一维数组*/
}

main()
{
    int a[3][4]={{1,3,5,2},{2,6,2,1},{3,0,8,9}};
    int b[3][4]={{5,6,8,9},{2,1,0,0},{6,9,2,3}};
    int r[3][4],i,j;
    MatrixAdd(a,b,r);
    for(i=0;i<3;i++)
    {
```



```
        for(j=0;j<4;j++)
            printf("%d ",r[i][j]);
        printf("\n");
    }
    getch();
}
```

第二种方法：应用指向二维数组每一行的指针。

```
#include <string.h>
#include <stdio.h>
MatrixAdd(int (*a)[4],int (*b)[4],int (*r)[4])
{
    int i,j;
    for(i=0;i<3;i++)
        for(j=0;j<4;j++) /*a,b,r 指向二维数组的每一行*/
            *(*r+i)+j)=*(*a+i)+j)+*(*b+i)+j);
}

main()
{
    int a[3][4]={1,3,5,2},{2,6,2,1},{3,0,8,9}};
    int b[3][4]={5,6,8,9},{2,1,0,0},{6,9,2,3}};
    int r[3][4],i,j;
    MatrixAdd(a,b,r);
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
            printf("%d ",r[i][j]);
        printf("\n");
    }
    getch();
}
```

两段代码的执行结果是一样的：

```
6 9 13 11
4 7 2 1
9 9 10 12
```

## 18.7 函数

函数在C语言中有着极其重要的地位。它是结构化程序设计的基础，是一个程序中实现某一特定功能的模块，本节将主要分析一些与函数相关的题目，主要包括：函数的定义、函数的参数、变量的类型、内部函数外部函数、函数的调用形式等相关知识。

### 例题 18-71

写出以下一段程序的执行结果。

```
void func(int v,int w)
{
    int t;
    t=v;v=w;w=t;
}
main()
```



```
{
    int x=1,y=3,z=2;
    if(x>y)
        func(x,y);
    else if(y>z) func(y,z);
    else func(x,z);
    printf("%d,%d,%d\n",x,y,z);
}
```

**分析:**

本题考查的是函数参数的传递方式以及变量的作用范围。根据第 4 章的介绍可以知道,函数参数的传递方式是值传递,也就是说实参与形参是不同的变量,函数调用时将实参的值传递给形参,以后被调函数对形参的操作不会影响到实参的值。另外,一旦被调函数执行完毕,被调函数中定义的所有变量(包括形参)将会被全部释放掉。

对于本题,调用函数 func 时,参数都是在主函数中定义的变量,因此函数 func 不会对主函数中的变量值产生任何影响, x, y, z 的值保持不变。

**答案:**

1, 3, 2

### 例题 18-72

以下一段程序的输出结果是 ( )

```
void swap1(int c[])
{
    int t;
    t=c[0];
    c[0]=c[1];
    c[1]=t;
}
void swap2(int c0,int c1)
{
    int t;
    t=c0;
    c0=c1;
    c1=t;
}
main()
{
    int a[2]={3,5},b[2]={3,5};
    swap1 (A) ;
    swap2(b[0],b[1]);
    printf("%d %d %d\n",a[0],a[1],b[0],b[1]);
}
```

(A) 5 3 5 3    (B) 5 3 3 5    (C) 3 5 3 5    (D) 3 5 5 3

**分析:**

本题考查的知识点是数组名作为函数的参数。当数组名作为函数的实参时,它就相当于一个地址,表示数组的首地址,因此形参也应该是指针变量或者如本例 int c[] 的形式,而函数对数组首地址的操作(指针操作)也会影响到数组本身。在本题中,函数 swap1 的参数是一个数组名,因此函数 swap1 的操作会直接影响到数组 a 的元素情况,也就是将 c[0] 和 c[1] 的内容对换;而函数 swap2 的参数只是数组元素,没有指针,因此同上例一样, swap2 不会影响到数组 b 的元素的值。



答案:

(B)

### 例题 18-73

以下一段程序的运行结果是 ( )

```
void fun(char *a,char *b)
{
    a=b;
    (*a)++;
}
main()
{
    char c1='A',c2='a',*p1,*p2;
    p1=&c1;
    p2=&c2;
    fun(p1,p2);
    printf("%c%c\n",c1,c2);
}
```

(A) Ab (B) aa (C) Aa (D) Bb

分析:

本题考查的知识点是指针作为函数的参数。本题中 p1, p2 定义为指向字符型的指针变量, c1, c2 为两个字符型变量, 并赋初值 'A', 'a'。然后将 p1 指向变量 c1, 将 p2 指向变量 c2, 这样 p1, p2 中就分别存放了 c1, c2 的地址, 将 p1, p2 作为参数传递给函数 fun。在函数 fun 中, 先将 b 赋值给 a, 也就是将 a 指向 c2, 然后将指针 a 指向的内容自增 1, 也就是将 c2 的值自增 1, 于是 c2 的值变为 b, c1 的值保持不变。

答案:

(A)

### 例题 18-74

写出以下一段程序的输出结果。

```
int *f(int *x,int *y)
{
    if(*x<*y)
        return x;
    else
        return y;
}
main()
{
    int a=7,b=8,*p,*q,*r;
    p=&a;
    q=&b;
    r=f(p,q);
    printf("%d,%d,%d\n ",*p,*q,*r);
}
```

分析:

本题考查的知识点是指针作为函数的参数。本题中 p, q, r 定义为指向整型的指针变量, a, b 定义为整型变量, 并赋初值 a=7 和 b=8。然后将 p 指向变量 a, 将 q 指向变量 b, 这样 p, q 中就分别存放了 a, b 的地址。然后将 p, q 作为参数传递给函数 f。函数 f 的返



返回值是一个指针，其作用是返回 a 和 b 中较小的那个变量的地址。就本题而言，函数 f 返回 a 的地址。然后将返回的地址赋值给指针变量 r，这样 r 也指向了变量 a。最后输出 p, q, r 分别指向的内容，也就是 a, b, a 的值。

**答案：**

7, 8, 7

#### 例题 18-75

全局变量和局部变量在内存中是否有区别？如果有，是什么区别？

**分析：**

本题考查的知识点是全局变量和局部变量在内存中的区别。要审清题目，问题是在内存中的区别，而不是在作用上的区别。根据第 4 章的介绍可以知道，全局变量储存在内存的静态数据区，而局部变量储存在堆栈上。当局部变量所处的函数调用完毕时，局部变量会被释放掉。也就是说局部变量的生命周期与它所处函数的生命周期相同；而全局变量的生命周期与整个程序的生命周期相同。

**答案：**

全局变量储存在内存的静态数据区，而局部变量储存在堆栈上。

#### 例题 18-76

局部变量和全局变量是否可以重名？

**分析：**

本题考查的知识点是局部变量和全局变量的命名规则。根据 C 语言的规定，局部变量可以与全局变量重名。在局部变量的作用域内，全局变量不起作用。

**答案：**

局部变量和全局变量可以重名。

#### 例题 18-77

简述 static 全局变量与普通的全局变量的区别。

**分析：**

根据第 4 章的介绍可以知道，外部变量（全局变量）是在函数外定义的变量，它不属于任何一个函数，源文件中的任何一个函数都可以使用。全局变量的内存空间分配在静态存储区中，作用域是从全局变量的定义点到源文件的结尾。全局变量可通过 extern 的外部说明为其他文件所引用，而加了关键字 static 的全局变量只限于在本文件中引用，而不能被其他文件引用。

**答案：**

普通的全局变量既可以被本文件中的函数所使用，又可以通过 extern 的外部说明被其他文件所引用；加了关键字 static 的全局变量只限于在本文件中引用，而不能被其他文件引用。

#### 例题 18-78

static 局部变量和普通局部变量有什么区别？



分析:

根据第4章的介绍可以知道,普通的局部变量都是自动变量,也就是说数据存储在动态存储区中。这类变量在调用函数时,由系统自动为它们分配内存空间,而当函数调用结束时,系统会自动释放这些空间。加了关键字 `static` 的局部变量则被定义为静态局部变量,它与自动变量不同,当函数调用结束时,它不会像自动变量那样被系统释放掉,而是保留原值,也就是说静态局部变量所占有的内存空间不因函数调用的结束而被释放。

答案:

普通的局部变量当函数调用结束时系统会自动释放它的内存空间;加了关键字 `static` 的局部变量在函数调用结束时系统不会自动释放它的内存空间,而是保留原值。

#### 例题 18-79

`static` 函数与普通函数有什么区别?

分析:

根据第4章的介绍可以知道,在C语言中,函数分为两种,一类函数既可以被本文件中的函数调用,又可以被其他文件中的函数调用,这类函数称为外部函数。还有一类函数只能被本文件中的其他函数调用,而不能被其他文件调用,这类函数称为内部函数。在函数的定义前加上关键字 `static` 就表明该函数是内部函数,它只能被本文件中的其他函数调用,而不能被其他文件的函数调用。

答案:

`static` 函数只能被本文件中的其他函数调用,而不能被其他文件的函数调用;普通函数既可以被本文件中的函数调用,又可以被其他文件中的函数调用。调用外部函数时,要加关键字 `extern` 说明。

#### 例题 18-80

写出下列代码的输出内容。

```
int a=5;
fun(int b)
{
    static int a=10;
    a+=b++;
    printf("%d",a);
}
main()
{
    int c=20;
    fun(c);
    a+=c++;
    printf("%d\n",a);
}
```

分析:

本题考查的知识点是函数的调用以及全局变量和静态局部变量。在本题中, `a` 定义为全局变量,并赋初值为5,在函数 `fun` 中又定义了同名的局部变量 `a`。根据C语言中的规定,在局部变量的作用范围内全局变量被屏蔽掉。因此当 `c=20` 作为参数传递给函数 `fun` 后, `a+=b++` 相当于把 `a` 加 `b` 的值赋值给 `a`,再做 `b` 自增1运算。这时 `a` 的值是30而不是25,然



后打印出 30。函数 fun 调用完毕后,再执行  $a+=c++$ ,相当于把 a 加 c 的值赋值给 a,再做 c 自增 1 运算。但是要注意,这时的 a 为全局变量的 a,因为主函数中没有定义与全局变量 a 同名的局部变量。因此 a 的值为 25,然后打印出 25。

答案:

30 25

### 例题 18-81

写出如下程序的输出结果。

```
long fib(int n)
{
    if(n>2)return (fib(n-1)+fib(n-2));
    else return 2;
}
main()
{
    printf("%d\n",fib(3));
}
```

分析:

本题考查的知识点是函数的递归调用。根据第 4 章的介绍可以知道,函数的递归调用是指一个函数在它的函数体内调用它自身。函数的递归调用是函数嵌套调用的一种特殊形式。在递归调用中,主调函数又是被调函数,这样自己调用自己,一层一层地调用下去。

本题中,主函数调用函数 fib,参数为 3,符合递归条件,再次调用函数 fib;而第二次调用该函数时参数分别为 2 和 1,满足结束条件,都返回 2;最后第一层的 fib 返回  $2+2=4$ 。所以程序的输出结果为 4。

答案:

4

### 例题 18-82

编写一个程序,用递归算法实现求一个数 n 的阶乘。

分析:

本题考查的知识点是函数的递归调用。阶乘运算本身就可以用递归的形式来定义:

$n!=1$  ( $n=0,1$ )

$n \times (n-1)!$  ( $n>1$ )

因此可以设计一个递归算法来实现阶乘的运算。

答案:

下面给出本题的代码清单。

```
#include <stdio.h>
long Fact(int n){ /*函数 Fact 实现阶乘运算,返回阶乘运算的结果*/
    if(n==0)
        return 1;
    else
        return n*Fact(n-1); /*递归地调用函数 Fact*/
}
int main(void)
{
```



```
int n;
long f;
printf("Please input a number:\n");
scanf("%d",&n);
f=Fact(n);
printf("Fact(%d)=%ld",n,f);
getchar();
return 0;
}
```

本程序的执行结果为：

```
Please input a number:
5
Fact(5)=120
```

## 18.8 结构与联合

结构与联合是C语言中的构造数据类型，它们是在基本数据类型（整型、字符型、浮点型、枚举类型）的基础上构造而成的更为复杂的数据类型。对于较为复杂的数据结构，应用结构与联合对其进行组织是很方便的。本节将介绍一些常见的有关结构与联合的题目。

### 例题 18-83

给定如下结构，求 sizeof(A) 的值。

```
struct A
{
    char t;
    char k;
    unsigned short i;
    unsigned long m;
};
```

分析：

本题考查的知识点是结构体占用空间的大小。根据第7章的介绍可以知道，定义了结构体变量，系统就要为该变量分配内存空间。系统为结构体变量分配空间的大小取决于结构体的成员变量所占空间的大小。对于本题，结构体类型A中有4个成员域，两个字符型变量t和k，一个无符号短整型变量i和一个无符号长整型变量m，因此该结构体类型所占的空间大小为：1+1+2+4=8（字节）。

答案：

8 字节

### 例题 18-84

写出下面一段程序执行后的输出结果。

```
struct STU
{
    char name[10];
    int num;
};
void f1(struct STU c)
{
```



```

    struct STU b={"Tom ",1001};
    c=b;
}
void f2(struct STU *c)
{
    struct STU b={"Lily",1002};
    *c=b;
}
main()
{
    struct STU a={"John",1003},b={"Bob",1004};
    f1(A) ;
    f2(&b);
    printf("%d,%d\n",a.num,b.num);
}

```

**分析:**

本题考查的知识点是结构体变量以及指向结构体变量的指针作为函数的参数。在本题中, a, b 都是结构体类型 struct STU 的变量, 并且给予了初始化。函数 f1 的参数是结构体变量本身, 根据函数参数的值传递原则, 函数 f1 的操作不会对结构体变量 a 产生任何影响。函数 f2 的参数是结构体变量的指针(地址), 因此函数 f2 的操作有可能影响到结构体变量 b 的值。在函数 f2 中, 将结构体{"Lily",1002}赋值给&b 所指向的内容, 其实就是赋值给结构体变量 b 本身, 因此结构体变量 b 的内容变为{"Lily",1002}。最后输出变量 a, b num 域的值。

**答案:**

1003, 1002

### 例题 18-85

写出下面一段程序的执行结果。

```

struct stu
{
    int num;
    char *name;
    char sex;
    float score;
} boy1={102,"Zhang ping",'M',78.5},*pstu;
main()
{
    pstu=&boy1;
    printf("Number=%d\nName=%s\n",boy1.num,boy1.name);
    printf("Sex=%c\nScore=%f\n\n",boy1.sex,boy1.score);
    printf("Number=%d\nName=%s\n",(*pstu).num,(*pstu).name);
    printf("Sex=%c\nScore=%f\n\n",(*pstu).sex,(*pstu).score);
    printf("Number=%d\nName=%s\n",pstu->num,pstu->name);
    printf("Sex=%c\nScore=%f\n\n",pstu->sex,pstu->score);
}

```

**分析:**

本题考查的知识点是结构体变量成员的引用。根据第7章的介绍可以知道, 引用结构体变量中的成员的方式大致分为两种, 一种是通过

变量名.成员名

方式引用结构体变量的成员。



另一种是通过

变量指针->成员名

方式引用结构体变量的成员。

本题就是针对这个知识点进行的考查。

答案:

```
Number=102
Name=Zhang ping
Sex=M
Score=78.500000

Number=102
Name=Zhang ping
Sex=M
Score=78.500000

Number=102
Name=Zhang ping
Sex=M
Score=78.500000
```

#### 例题 18-86

下面一段程序的输出结果为 ( )

```
main()
{
    struct cmplx{int x;int y;}cnum[2]={1,3,2,7};
    printf("%d\n",cnum[0].y/cnum[0].x*cnum[1].x);
}
```

(A) 0 (B) 1 (C) 3 (D) 6

分析:

本题考查的知识点是结构体数组的初始化以及结构体成员变量的引用。在本题中,定义了一个结构体数组 `cnum[2]`, 并为它初始化, 这样, `cnum[0].x=1`, `cnum[0].y=3`, `cnum[1].x=2`, `cnum[1].y=7`。然后按照算术表达式的运算顺序即可计算出结果:  $3/1*2=6$ 。

答案:

(D)

#### 例题 18-87

下面一段程序的运行结果为 ( )

```
typedef union
{
    long x[2];
    int y[4];
    char z[8];
} MYTYPE;
MYTYPE them;
main()
{
    printf("%d\n",sizeof(them));
}
```

(A) 32 (B) 16 (C) 8 (D) 24



分析:

本题考查的知识点是联合体占用内存的大小以及利用 typedef 来定义类型的相关知识。程序中首先将联合:

```
union
{
    long x[2];
    int y[4];
    char z[8];
}
```

定义为 MYTYPE 类型, 然后用 MYTYPE 声明了一个变量 them, 最后打印出 them 占用内存的字节数。根据第 7 章的介绍可以知道, 联合又叫作共用体, 采用的是覆盖技术, 因此联合类型变量的大小应为联合中最长成员的长度。本题中长整型 long 占用 4 字节, 整型 int 占用 2 字节, 字符型 char 占用 1 字节, 则整个联合体变量 them 所占的内存空间大小应该是其中占用空间最多的成员大小。长整型占用  $4 \times 2 = 8$  字节; 整型占用  $2 \times 4 = 8$  字节; 字符型占用  $1 \times 8 = 8$  字节, 三个成员占用内存大小相等, 所以 `sizeof(them)=8`。

答案:

(C)

#### 例题 18-88

下面一段程序的输出结果为 ( )

```
main()
{
    union
    {
        int k;
        char i[2];
    } *s, a;
    s = &a;
    s->i[0] = 0x39; s->i[1] = 0x38;
    printf("%x\n", s->k);
}
```

(A) 3839 (B) 3938 (C) 380039 (D) 390038

分析:

本题考查的知识点是联合体共同使用内存的特点。本题中联合共占用 2 字节的内存空间, k 和数组 i 使用同一个内存空间。因此在向数组 i 赋值时, 也就等于向整型变量 k 的低、高字节分别赋值, i[0] 为 k 的低字节, i[1] 为 k 的高字节。本题中的数据是以十六进制的形式表示的, 输出时仍按十六进制形式输出, 因此输出结果为 3839。

答案:

(A)

## 18.9 位运算

C 语言是介于高级语言与低级语言之间的一门语言, 这主要是因为 C 语言具有强大的



对位操作的功能,使得C语言可以进行一些更为底层的操作。本节将介绍一些常见的有关位运算的题目。

### 例题 18-89

有下面一段代码,请问 hash(16), hash(256)的值分别是多少?

```
unsigned short hash(unsigned short key)
{
    return (key>>1)%256;
}
```

分析:

本题考查的知识点是位运算中的右移运算。关键是理解程序中 `key>>1` 的作用。`key>>1` 的作用是将 `key` 值的二进制数位右移一位,当 `key` 等于 16 时,它的二进制表示为 10000,右移一位得 01000,也就是十进制的 8,再与 256 进行模运算,最后得 8;当 `key` 等于 256 时,它的二进制表示为 100000000,右移一位得 010000000,也就是十进制的 128,再与 256 进行模运算,最后得 128。

答案:

hash(16)=8, hash(256)=128。

### 例题 18-90

编写一段程序,要求不使用第三变量交换两个变量的值。

分析:

本题考查的知识点是位运算的灵活运用。在位运算中,可以应用位操作实现许多特殊的功能。例如交换两个变量的值,一般的方法是通过一个中间变量实现,而如果巧妙地利用位运算,就可以省掉中间变量。

设变量 `a`, `b`, 现在要交换变量 `a`, `b` 的值,则可以通过

```
a=a^b;
b=b^a;
a=a^b;
```

的一组位运算来实现,其中`^`为异或运算符。具体内容可参看 8.2.3 节。

答案:

下面给出本题的代码清单。

```
main()
{
    int a, b;
    scanf("%d%d",&a,&b);
    a=a^b;
    b=b^a;
    a=a^b;
    printf("%d %d",a,b)
}
```

### 例题 18-91

有下面一段程序,则 `b` 的值是 ( )

```
main()
```



```
{
    int x=35;
    char z='A';
    int b;
    b=((x&15)&&(z<'a'));
}
```

(A) 0 (B) 1 (C) 2 (D) 3

分析:

本题考查的知识点是位运算中的按位与运算。首先要会计算  $35 \& 15$ ，它相当于:

$$\begin{array}{r} 100011 \\ \& 001111 \\ \hline 000011 \end{array}$$

值等于 3。又由于 'A' 的 ASCII 码为 65，'a' 的 ASCII 码为 97，所以  $z < 'a'$  的值为真。逻辑表达式  $(x \& 15) \&& (z < 'a')$  的值为 1。

答案:

(B)

#### 例题 18-92

下面一段程序的输出结果为 ( )

```
main()
{
    unsigned char a,b;
    a=4|3;
    b=4&3;
    printf("%d,%d\n",a,b);
}
```

(A) 7, 0 (B) 0, 7 (C) 1, 1 (D) 43, 0

分析:

本题考查的知识点是位运算中的按位或运算和按位与运算。 $4|3=100_2|011_2=111_2=7$ ;  
 $4 \& 3=100_2 \& 011_2=000_2=0$ 。因此 a 的值为 7，b 的值为 0。

答案:

(A)

#### 例题 18-93

写出下面一段程序的输出结果。

```
typedef struct
{
    int a:2;
    int b:2;
    int c:1;
}test;
main()
{
    test t;
    t.a = 1;
    t.b = 3;
    t.c = 1;
```



```
printf("%d",t.a);
printf("%d",t.b);
printf("%d",t.c);
}
```

**分析:**

本题考查的知识点是位段。根据第8章的介绍可知,位段定义的一般形式为:

```
struct 位段结构名
{ 位段列表 };
```

其中位段列表的形式为:

```
类型说明符 位段名: 位段长度
```

本题中应用 typedef 将位段定义为 test 类型,然后定义了一个位段变量 t,并将 3 个域分别赋值为 1, 3, 1。其内部的二进制存储形式分别为 01, 11, 1 (因为位段长度分别为 2, 2, 1)。当作为整数输出这三个位段时,将 01 的前面补 0, 输出为 1, 将 11 的前面补 1, 输出为 -1, 将 1 的前面补 1, 输出为 -1。(计算机内部以二进制补码形式存储数据)

**答案:**

1-1-1

#### 例题 18-94

写出以下程序的输出结果。

```
main()
{
    unsigned t=129;
    t=t^00;
    printf("%d,%o",t,t);
}
```

**分析:**

本题考查的知识点是位运算中的异或运算。程序中 t 被定义为无符号整数,且赋初值为 129, 其二进制表示为 10000001, 然后与 0 进行异或运算。异或运算的规则是相同的位运算结果为 0, 不同的位运算结果为 1。因此 t 与 0 的异或运算为  $10000001 \wedge 00000000 = 10000001$ 。二进制数 10000001 的十进制表示为 129, 八进制表示为 201, 因此程序的输出结果为 129, 201。

**答案:**

129, 201

#### 例题 18-95

有以下语句, 则 c 的二进制值是 ( )

```
char a=3,b=6,c;
c=a^b<<2;
```

(A) 00011011    (B) 00010100    (C) 00011100    (D) 00011000

**分析:**

本题考查的知识点是位运算的混合运算。在位运算中, 左移或右移运算的优先级大于其他二目运算, 因此在这里要先算  $b \ll 2$ , 得到二进制数 00011000, 再与 a 进行异或运算,



最终得到二进制数 00011011。

答案:

(A)

## 18.10 文件操作

“文件”是指一组相关数据的有序集合，在 C 程序设计中具有重要作用。操作系统以文件为单位对计算机内的数据进行管理，在开发一些较大的应用程序时，程序中的数据可以以文件的形式存储在外部设备上，以减轻程序的负载。本节将对一些与文件操作相关的题目加以分析。

### 例题 18-96

简述二进制文件与 ASCII 文件的区别。

分析:

从文件编码的方式来看，文件可分为 ASCII 码文件和二进制码文件两种。

ASCII 文件又叫做文本文件，它在磁盘上的存储形式是每个字符对应一字节，字符以对应的 ASCII 码形式存储。例如源程序文件就是 ASCII 文件，ASCII 文件可以在终端显示出来，因为它们都是可打印的字符。

二进制文件是按二进制的编码方式来存放文件的。由于一些数据的编码可能是不可打印字符的计算机内部码，因此二进制文件一般无法读懂。

二进制文件与 ASCII 文件只是人为的划分，其实系统在处理这些文件时，并不区分类型，都看成是字符流，按字节进行处理。

答案:

见分析。

### 例题 18-97

以下程序运行后的输出结果是 ( )

```
#include<stdio.h>
main()
{
    FILE *fp;
    int i=20,j=30,k,n;
    fp=fopen("d1.dat","w");
    fprintf(fp, "%d\n",i);
    fprintf(fp, "%d\n",j);
    fclose(fp);
    fp=fopen("d1.dat","r");
    fscanf(fp, "%d%d",&k,&n);
    printf("%d%d\n",k,n);
    fclose(fp);
}
```

(A) 20 30    (B) 20 50    (C) 30 50    (D) 30 20



分析:

本题考查的知识点是用于文件操作的相关函数。题目中涉及的文件操作函数包括: 文件打开函数 `fopen`、文件写函数 `fprintf`、文件读函数 `fscanf` 以及文件关闭函数 `fclose`。程序首先以写方式打开一个名为“d1.dat”的文件, 然后将整数 `i=20`, `j=30` 写入文件, 最后关闭文件。再以读方式打开该文件, 并将文件中事先存入的数读到变量 `k` 和 `n` 中, 并打印出来。所以输出结果为 20 30。

答案:

20 30

#### 例题 18-98

以下程序段打开文件后, 先利用 `fseek` 函数将文件位置指针定位在文件末尾, 然后调用 `ftell` 函数返回当前文件位置指针的具体位置, 从而确定文件的长度。请填空。

```
main()
{
    FILE *myf;
    long f;
    myf=_____("test.txt","rb");
    fseek(myf,0,SEEK_END);
    f=ftell(myf);
    fclose(myf);
    printf("%d\n",f);
}
```

分析:

本题考查的是对文件进行操作的步骤。要对一个指定的文件进行操作, 首先要将它从外存读到内存的缓冲区中, 然后才能应用其他的文件操作函数(例如 `fread`、`fwrite`、`fprintf`、`fseek` 等)对文件进行读写、定位等操作。函数 `fopen` 的作用就是将文件从外存读到内存缓冲区中, 并将内存缓冲区的指针(`FILE` 类型)返回, 只有通过这个指针, 其他的文件处理函数才能对文件进行操作, 因此这里缺少文件打开函数 `fopen`。

答案:

`fopen`

#### 例题 18-99

请简述 `stdin`, `stdout`, `stderr` 的含义。

分析:

在操作系统中, I/O 设备都是用文件进行管理的, 因此设备都配有相应的控制文件。在 C 语言中, 有 3 个文件与终端相联系, 被称作标准文件指针, 它们分别是: 标准输入指针 `stdin`, 标准输出指针 `stdout`, 标准出错输出指针 `stderr`。通过这 3 个指针可以向终端输出数据或错误信息, 以及从终端输入数据。

答案:

见分析。

#### 例题 18-100

编写一个 C 程序, 用来计算指定文件的大小。



分析:

计算指定文件大小的方法很多。最直观的方法是通过扫描整个文件计算出文件的字节数,但是这种方法对系统的开销很大,比较浪费时间。例题 18-98 的程序就是一个计算文件大小的较为优化的程序,程序中用到了两个重要的函数 `fseek` 和 `ftell`, 它们的功能如下。

`fseek(FILE *fp, long offset, int base)`: 重定位流上的文件指针, 即将 `fp` 指向的文件的位置指针移向以 `base` 为基准, 以 `offset` 为偏移量的位置。

`ftell(FILE *fp)`: 返回当前文件指针的位置。这个位置是指当前文件指针相对于文件开头的位移量。

因此可以先通过函数 `fseek` 将文件的指针定位到文件的最后 `SEEK_END`, 然后通过函数 `ftell` 返回当前文件指针相对于文件开头的位移量, 也就是文件的长度。

答案:

```
main()
{
    FILE *myf;
    long f;
    myf=fopen("test.txt","rb");
    fseek(myf,0,SEEK_END);
    f=ftell(myf);
    fclose(myf);
    printf("%d\n",f);
}
```